- 以云计算与大数据融合的视角阐述了云计算环境下Spark大数据处理与相应的算法实现
- 结合经典案例，详解云计算环境下Spark大数据处理生态圈，包括系统结构、大数据存储、批处理、流计算、交互式数据分析、并行机器学习架构与算法等技术
- 掌握云计算环境下Spark大数据处理的架构搭建和算法实现过程等关键技术，扩展大数据从业人员的理论与实践能力





Lightning-fast Cluster Computing

云计算环境下
Spark大数据处理技术与实践

邓立国　佟强　著

清华大学出版社



云计算环境下Spark大数据处理技术与实践

邓立国　佟强　著

清華大学出版社
北　京

内 容 简 介

本书围绕互联网重大的技术革命：云计算、大数据进行阐述。云计算环境下大数据处理构建是国民经济发展的信息基础设施，发展自主的云计算核心技术，拥有自己的信息基础设施，当前正处于重要的机遇期。

本书重点在大数据与云计算的融合，给出了大数据与云计算的一些基本概念，并以 Spark 为开发工具，全面讲述云环境下的 Spark 大数据技术部署与典型案例算法实现，最后介绍了国内经典 Spark 大数据与云计算融合的架构与算法。

本书适合云计算环境下 Spark 大数据技术人员、Spark MLlib 机器学习技术人员，也适合高等院校和培训机构相关专业的师生教学参考。

图书在版编目（CIP）数据

云计算环境下 Spark 大数据处理技术与实践 / 邓立国，佟强著. — 北京：清华大学出版社，2017

ISBN 978-7-302-47971-0

I. ①云… II. ①邓… ②佟… III. ①数据处理软件 IV. ①TP274

中国版本图书馆 CIP 数据核字（2017）第 207679 号

责任编辑：夏毓彦
封面设计：王　翔
责任校对：闫秀华
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张：**22.25　　**字　　数：**570 千字
版　　次：2017 年 9 月第 1 版　　**印　　次：**2017 年 9 月第 1 次印刷
印　　数：1~3500
定　　价：69.00 元

产品编号：075719-01

前　言

麦肯锡全球研究所给出的大数据定义是：一种规模大到在获取、存储、管理、分析方面大大超出了传统数据库软件工具能力范围的数据集合，具有海量的数据规模、快速的数据流转、多样的数据类型和价值密度低四大特征。

大数据技术的战略意义不在于获取了庞大的数据，而在于对这些特定领域的数据进行处理分析。换而言之，关键是把这些巨大的数据实现盈利式的加工，提供效率，具有增值的处理模式。

本书背景

大数据像飓风一样席卷而来，改变着信息时代的数据处理方式。产业经营方式经历着革命性的变革，大数据与云计算的融合改变着数据处理流程和模式，对互联网、信息经济发展提出了新的方向和扩展空间。应用驱动技术发展产生的数据越多，可供分析的数据越多，越能推动研发和出现更先进的用来分析数据的工具和方法。

国家对互联网、信息经济的发展提出了方向，明确说要拓展发展新的空间，实施网络强国战略，实施“互联网+”行动计划，发展分享经济，实施国家大数据战略，将网络强国战略作为新的一个创新的重要支撑。

本书内容

本书围绕互联网重大的技术革命：云计算、大数据（未来世界新一代信息技术的关键和核心）进行阐述。云计算环境下大数据处理构建是国民经济发展的信息基础设施，发展自主的云计算核心技术，拥有自己的信息基础设施，当前正处于重要的发展机遇期。本书重点在大数据与云计算的融合，给出了大数据与云计算的一些基本概念的同时，以 Spark 为开发工具，全面讲述云环境下的大数据技术部署与典型案例算法实现，最后介绍了国内经典 Spark 大数据与云计算融合的架构与算法。

本书目的

3 年前就开始着手准备写关于大数据和云计算融合的相关技术方面的书，由于书中的算法需要模拟验证，所以交稿拖延了很长时间。目前这方面的书还不系统，还没有全面融合两

者技术的书出现，也是笔者想写这本书的初衷。随着岁月侵蚀，白发杂生，大数据技术发展也日新月异。

得益于国内 IT 企业的后发制人战略，目前国内的 IT 公司在大数据应用方面已经迎头赶上了国际巨头，在云大数据技术方面的研发和技术突破经历了大幅的跨越发展。当今世界迎来大数据时代，工欲善其事，必先利其器，在大数据和云计算的规则制定和新技术研发上还需努力，这方面还需要加大研发与突破。

致谢

感谢家人给我的全身心的支持与关爱，没有你们的宽容与支持即使是 10 年也没法完成这本书。由于撰写时间紧迫，夜晚孤灯，每晚多想陪着妻子月夜树影婆娑，多想在闺女的校门口等待闺女背着书包颠颠地跑来。最后感谢单位给予的大力支持与帮助。

著者

2017 年 8 月

目　录

第 1 章

◂ 大数据处理概述 ▸

大数据是当今一个最热门的话题，我们每一个人都无法置身其外。就像几年前出现的云计算一样，大数据已经引起市场的广泛关注；同样，企业迫切需要对大数据下定义。大数据缺少一个标准且普及性的定义，至少不像 NIST 对云的定义那样，能被人们广泛接受。调研公司 IDC 的定义可能比较容易被人们所接受。它对大数据的定义是：一种新一代的技术和架构，具备高效率的捕捉、发现和分析能力，能够经济地从类型繁杂、数量庞大的数据中挖掘出价值。

1.1 大数据处理技术概述

近几年，大数据迅速发展成为科技界和企业界甚至世界各国政府关注的热点。《Nature》和《Science》等相继出版专刊专门探讨大数据带来的机遇和挑战。著名管理咨询公司麦肯锡称：“数据已经渗透到当今每一个行业和业务职能领域，成为重要的生产因素。人们对于大数据的挖掘和运用，预示着新一波生产力增长和消费盈余浪潮的到来”。美国政府认为大数据是“未来的新石油，一个国家拥有数据的规模和运用数据的能力将成为综合国力的重要组成部分，对数据的占有和控制将成为国家间和企业间新的争夺焦点。大数据已成为社会各界关注的新焦点，“大数据时代”已然来临[1]。

“大数据”是一个体量特别大、数据类别特别大的数据集，并且这样的数据集无法用传统数据库工具对其内容进行抓取、管理和处理。

百度知道大数据（bigdata）的定义，或称巨量资料，指的是所涉及的资料量规模巨大到无法透过目前主流软件工具，在合理时间内达到撷取、管理、处理，并整理成为帮助企业经营决策更积极目的的资讯。大数据的 5V 特点：Volume、Velocity、Variety、Veracity、Value。

1.1.1 什么是大数据

“大数据”是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力的海量、高增长率和多样化的信息资产。从数据的类别上看，“大数据”指的是无法使用传统流程或工具处理或分析的信息。它定义了那些超出正常处理范围和大小、迫使用户采用非传统

处理方法的数据集。亚马逊网络服务（AWS）大数据科学家 JohnRauser 提到一个简单的定义：大数据就是任何超过了一台计算机处理能力的庞大数据量。其研发小组对大数据的定义："大数据是最大的、最时髦的技术，当这种现象出现时，定义就变得很混乱。"学者 Kelly 说："大数据是可能不包含所有的信息，但我觉得大部分是正确的。对大数据的一部分认知在于，它是如此之大，分析它需要多个工作负载，这是 AWS 的定义。当你的技术达到极限时，也就是数据的极限"。大数据不是关于如何定义，最重要的是如何使用。最大的挑战在于哪些技术能更好地使用数据以及大数据的应用情况如何。这与传统的数据库相比，开源的大数据分析工具如 Hadoop 的崛起，这些非结构化的数据服务的价值在哪里。

相较于传统的数据，人们将大数据的特征总结为 5 个 V，即体量大（Volume）、速度快（Velocity）、模态多（Variety）、难辨识（Veracity）和价值大（Value）。"大数据"首先是指数据体量(volumes)大，指代大型数据集，一般在 10TB 规模左右，但在实际应用中，很多企业用户把多个数据集放在一起，已经形成了 PB 级的数据量；其次是指数据类别（Variety）多，数据来自多种数据源，数据种类和格式日渐丰富，已冲破了以前所限定的结构化数据范畴，囊括了半结构化和非结构化数据；接着是数据处理速度（Velocity）快，在数据量非常庞大的情况下，也能够做到数据的实时处理；还有一个特点是指数据真实性（Veracity）高，随着社交数据、企业内容、交易与应用数据等新数据源的兴趣，传统数据源的局限被打破，企业愈发需要有效的信息之力以确保其真实性及安全性。但大数据的主要难点并不在于数据量大，因为通过对计算机系统的扩展可以在一定程度上缓解数据量大带来的挑战。其实，大数据真正难以对付的挑战来自于数据类型多样（Variety）、要求及时响应（Velocity）和数据的不确定性（Veracity）。因为数据类型多样使得一个应用往往既要处理结构化数据，同时还要处理文本、视频、语音等非结构化数据，这对现有数据库系统来说难以应付；在快速响应方面，在许多应用中时间就是利益；在不确定性方面，数据真伪难辨是大数据应用的最大挑战。追求高数据质量是对大数据的一项重要要求，最好的数据清理方法也难以消除某些数据固有的不可预测性。

1.1.2 大数据来源

当今世界，大数据无处不在，它影响到了我们的工作、生活和学习，并将继续施加更大的影响。大数据用于描述这样的数据组，其规模超出了日常软件在可容忍期限内获取、管理和加工数据的能力。一些网络技术领先的公司持续地投资于昂贵的大数据技术，成效显著。大数据使得创新型公司变成了经营新方法的率先接受者，经营更为成功。通过大数据的分析挖掘，公司可以发现新的经营模式，对工艺加以改进。例如，在获悉消费者行为后，可以将发现用于某些改变，如降低成本或增加销售，就会产生价值。在任意大的数据组中应用统计方法可以发现有用信息，将这些信息商业化即可获益。

当今大数据的来源除了专业研究机构产生大量的数据外（CERN 的离子对撞机每秒运行产生的数据高达 40TB），相对于企业，大数据的数据来源主要有两个部分：一部分来自于企业内部自身信息系统中产生的运营数据，这些数据大多是标准化、结构化的；传统的商业智能系统中所用到的数据基本上属于这部分；另一部分则来自于外部，包括广泛存在于社交网络、物联网、电子商务等之中的非结构化数据。

与企业经营相关的大数据可以划分为 4 个来源：

- 越来越多的机器配备了连续测量和报告运行情况的装置。几年前，跟踪遥测发动机运行仅限于价值数百万美元的航天飞机。现在，汽车生产商在车辆中配置了监视器，连续提供车辆机械系统整体运行情况。一旦数据可得，公司将千方百计从中渔利。这些机器传感数据属于大数据的范围。
- 计算机产生的数据可能包含着关于因特网和其他使用者行动和行为的有趣信息，从而提供了对他们的愿望和需求潜在的有用认识。
- 使用者自身产生的数据/信息。人们通过电邮、短信、微博等产生的文本信息。
- 至今最大的数据是音频、视频和符号数据。这些数据结构松散、数量巨大，很难从中挖掘有意义的结论和有用的信息。

由于来源、类型不同的数据透视的是同一个事物的不同方面，以消费客户为例，消费记录信息能透视客户的消费能力、消费频率、消费兴趣点等，渠道信息能透视客户的渠道偏好，以及消费支付信息与支付渠道的关联情况等。

大型以 Internet 为核心的公司，如 Amazon、Google、eBay、Twitter 和 Facebook 正使用后三类海量信息认识消费行为、预测特定需求和整体趋势。第一类数据可能产生较少的业务，但可以推动某些经营模式实质变革。例如，汽车传感数据用于评价司机行为会推动汽车保险业的深刻变革。因此，大数据分析意味着企业能够从不同来源的数据中获得新的洞察力，并将其与企业业务体系的各个细节相结合，以便助力企业在市场拓展和产品创新上突破。

1.1.3 大数据应用价值

“大数据”的概念远不止大量的数据（TB）和处理大量数据的技术，而是涵盖了人们在大规模数据的基础上可以做的事情，而这些事情在小规模数据的基础上是无法实现的。换句话说，大数据让我们以一种前所未有的方式，通过对海量数据进行分析，获得有巨大价值的产品和服务，或深刻的洞见，最终形成变革之力。

根据麦肯锡全球研究所的分析，利用大数据在各行各业能产生显著的财务价值。美国健康护理利用大数据每年产出 3000 亿美元，年劳动生产率提高 0.7%；欧洲公共管理每年价值 2500 亿欧元，年劳动生产率提高 0.5%；全球个人定位数据服务提供商收益 1000 多亿美元，为终端用户提供高达 7000 亿美元的价值；美国零售业净收益可增长 6%，年劳动生产率提高 0.5~1%；制造业可节省 50%的产品开发和装配成本，营运资本下降 7%。

大数据改变了所有行业全部公司的经营方式。从对市场的理解到如何挖掘经营信息，大数据能洞察每项转变。一个致力于收集和分析大数据的行业业已形成，对现有公司产生了深刻影响。据有关调查，有 10%的公司认为在过去的 5 年中，大数据彻底改变了它们的经营方式。46%的公司认同大数据是其决策的一项重要支持因素。

大数据应用在经历了喊口号、布局深耕之后，开始显现出巨大的商业价值，触角延伸到国防、市政、金融、教育、医疗、体育、汽车、影视、智能硬件、社交网络等各个层面。据 IDC 数据显示，目前大数据形成的市场规模达到约 51 亿美元，到 2017 年，这一数字将会增

长到 530 亿美元。

近日，国内大数据精准营销平台亿玛公司联合中关村大数据产业联盟举办了 2015 亿玛智慧峰会，智能穿戴设备、智能汽车、互联网金融、大数据精准营销、智能家居、大数据医疗与健康、移动智能大数据应用等大数据应用代表企业共同围绕“大数据，智未来”的主题，讨论了 “如何依托大数据提供更符合社会需求的产品和服务”“大数据如何为精准营销提供强大的驱动力”等业界最关注的热点话题。

截至目前，大数据应用的商业价值已经在互联网金融、智能可穿戴设备、人工智能、智慧城市、精准营销等多个领域体现。其中，在互联网金融领域，通过分析大量的网络交易及行为数据，可对用户进行信用评估，从而帮助互联网金融企业对用户还款意愿及还款能力得出结论，继而为用户提供快速授信及现金分期服务。

而智能可穿戴设备“真正”的主线产品是由云端大数据引出的软件与服务。将来在新的模式里，硬件和服务都要具备快速迭代的能力，同时集成更多的传感器，数据来源将越来越丰富，而数据分析服务将利用更多种类的数据来交叉分析，无须用户干预而通过数据智能化学习就能把人一天重要的生理活动描绘出来，通过数据把行为量化。

大数据一个主要特性是复杂，这就意味着它的多元性。大数据不再是结构化数据，因此针对数据分析的模型和理论都必须重新构建，甚至分析大数据行为特征所依托的软硬件都必须进行变革。

1.1.4 大数据技术特点和研究内容

根据国际数据公司（IDC）的测算，2011 年数字世界将产生 1800EB 的数据，2012 年会增长 40%，达到 2500EB。截止 2020 年，会达到 35000EB，似乎没有足够的磁盘空间存储。就传统 IT 企业来看，其结构化和非结构化的数据增长也是惊人的。2005 年企业存储的结构化数据为 4EB，到 2015 年将增至 29EB，年复合增长率逾 20%。非结构化数据发展更猛。2005 年为 22EB，2015 年将增至 1600EB，年复合增长率约 60%，远远快于摩尔定律。

大数据具有 5 个主要的技术特点，人们将其总结为 5V 特征：

- Volume（大体量）：可从数百 TB 到数十数百 PB，甚至 EB 的规模。
- Variety（多样性）：大数据包括各种格式和形态的数据。
- Velocity（时效性）：很多大数据需要在一定的时间限度下得到及时处理。
- Veracity（准确性）：处理的结果要保证一定的准确性。
- Value（大价值）：大数据包含很多深度的价值，大数据分析挖掘和利用将带来巨大的商业价值。

传统的数据库系统主要面向结构化数据的存储和处理，但现实世界中的大数据具有各种不同的格式和形态。据统计，现实世界中 80%以上的数据都是文本和媒体等非结构化数据；同时，大数据还具有很多不同的计算特征。我们可以从多个角度分类大数据的类型和计算特征：

- 从数据结构特征角度看，大数据可分为结构化与非结构化/半结构化数据。
- 从数据获取处理方式看，大数据可分为批处理与流式计算方式。

- 从数据处理类型看，大数据处理可分为传统的查询分析计算和复杂数据挖掘计算。
- 从大数据处理响应性能看，大数据处理可分为实时/准实时与非实时计算，或者是联机计算与线下计算。前述的流式计算通常属于实时计算，此外查询分析类计算通常也要求具有高响应性能，因而也可以归为实时或准实时计算。而批处理计算和复杂数据挖掘计算通常属于非实时或线下计算。
- 从数据关系角度看，大数据可分为简单关系数据（如 Web 日志）和复杂关系数据（如社会网络等具有复杂数据关系的图计算）。
- 从迭代计算角度看，现实世界的数据处理中有很多计算问题需要大量的迭代计算，诸如一些机器学习等复杂的计算任务会需要大量的迭代计算，为此需要提供具有高效的迭代计算能力的大数据处理和计算方法。
- 从并行计算体系结构特征角度看，由于需要支持大规模数据的存储和计算，因此目前绝大多数大数据处理都使用基于集群的分布式存储与并行计算体系结构和硬件平台。MapReduce 是最为成功的分布式存储和并行计算模式。然而，基于磁盘的数据存储和计算模式使 MapReduce 难以实现高响应性能。为此人们从分布计算体系结构层面上又提出了内存计算的概念和技术方法。

大数据的研究与分析应用的意义和价值十分重大，带来巨大的挑战、技术创新与商机。维克托·迈克-舍恩伯格在《大数据时代》中列举了大量翔实的案例，指出了大数据的发展思路，大数据开启了生活、工作和创新的思维模式，影响了我们的经济、政治、科技和社会发展的各个领域，由于大数据应用行业需求的日益增长，大数据的并行计算技术越来越多地渗透到每个涉及大规模数据和复杂计算的应用领域。因此，以大数据处理为中心的技术变革，直接刺激计算机体系结构、操作系统、数据库、编译技术、程序设计、软件工程、多媒体信息处理、人工智能以及其他计算机应用技术，融合传统技术产生很多相应的新的研究课题与热点。

1.1.5　大数据计算与系统

大数据中蕴含的宝贵价值成为人们存储和处理大数据的驱动力。维克托·迈克-舍恩伯格在《大数据时代》一书中指出了大数据时代处理数据理念的三大转变，即要全体不要抽样，要效率不要绝对精确，要相关不要因果。因此，海量数据的处理对于当前存在的技术来说是一种极大的挑战。目前，人们对大数据的处理形式主要是对静态数据的批量处理、对在线数据的实时处理，以及对图数据的综合处理。其中，在线数据的实时处理又包括对流式数据的处理和实时交互计算两种。

MapReduce 计算模式的出现有力地推动了大数据技术和应用的发展，使其成为目前大数据处理最成功的主流大数据计算模式。然而，现实世界中的大数据处理问题复杂多样，难以有一种单一的计算模式能涵盖所有不同的大数据计算需求。研究和实际应用中发现，由于 MapReduce 主要适合于进行大数据线下批处理，在面向低延迟和具有复杂数据关系和复杂计算的大数据问题时有很大的不适应性。因此，近几年来学术界和业界在不断研究并推出多种不同的大数据计算模式。所谓大数据计算模式，是指根据大数据的不同数据特征和计算特征，从多样性的大数据计算问题和需求中提炼并建立的各种高层抽象（Abstraction）和模型

（Model）。传统的并行计算方法主要从体系结构和编程语言的层面定义了一些较为底层的抽象和模型，但由于大数据处理问题具有很多高层的数据特征和计算特征，因此大数据处理需要更多地结合其数据特征和计算特性考虑更为高层的计算模式[3]。

根据大数据处理多样性的需求，目前出现了多种典型和重要的大数据计算模式与系统。与这些计算模式相适应，出现了很多对应的大数据计算模式和系统工具。本节将详细阐述上述各种数据形式的特征和各自的典型应用以及相应的代表性系统。

1. 大数据查询分析计算 HBase、Hive、Cassandra、Premel、Impala、Shark、Hana、Redis 等

大数据查询分析是云计算中的核心问题之一，Google 在 2006 年之前的几篇论文奠定云计算领域基础，尤其是 GFS、Map-Reduce、Bigtable 被称为云计算底层技术的三大基石。GFS、Map-Reduce 技术直接支持了 Apache Hadoop 项目的诞生。Bigtable 和 Amazon Dynamo 直接催生了 NoSQL 这个崭新的数据库领域，撼动了 RDBMS 在商用数据库和数据仓库方面几十年的统治性地位。FaceBook 的 Hive 项目是建立在 Hadoop 上的数据仓库基础构架，提供了一系列用于存储、查询和分析大规模数据的工具。当我们还沉浸在 GFS、Map-Reduce、Bigtable 等 Google 技术中，并进行理解、掌握、模仿时，Google 连续推出了多项新技术，包括：Dremel、Pregel、Percolator、Spanner 和 F1。其中，Dremel 促使了实时计算系统的兴起，Pregel 开辟了图数据计算这个新方向，Percolator 使分布式增量索引更新成为文本检索领域的新标准，Spanner 和 F1 向我们展现了跨数据中心数据库的可能。在 Google 的第二波技术浪潮中，基于 Hive 和 Dremel，新兴的大数据公司 Cloudera 开源了大数据查询分析引擎 Impala，Hortonworks 开源了 Stinger，Fackbook 开源了 Presto。类似 Pregel，UC Berkeley AMPLAB 实验室开发了 Spark 图计算框架，并以 Spark 为核心开源了大数据查询分析引擎 Shark。由于某电信运营商项目中大数据查询引擎选型需求，本节将会对 Hive、Impala、Shark、Stinger 和 Presto 这五类主流的开源大数据查询分析引擎进行简要介绍以及性能比较。Hive、Impala、Shark、Stinger 和 Presto 的进化图谱如图 1-1 所示。

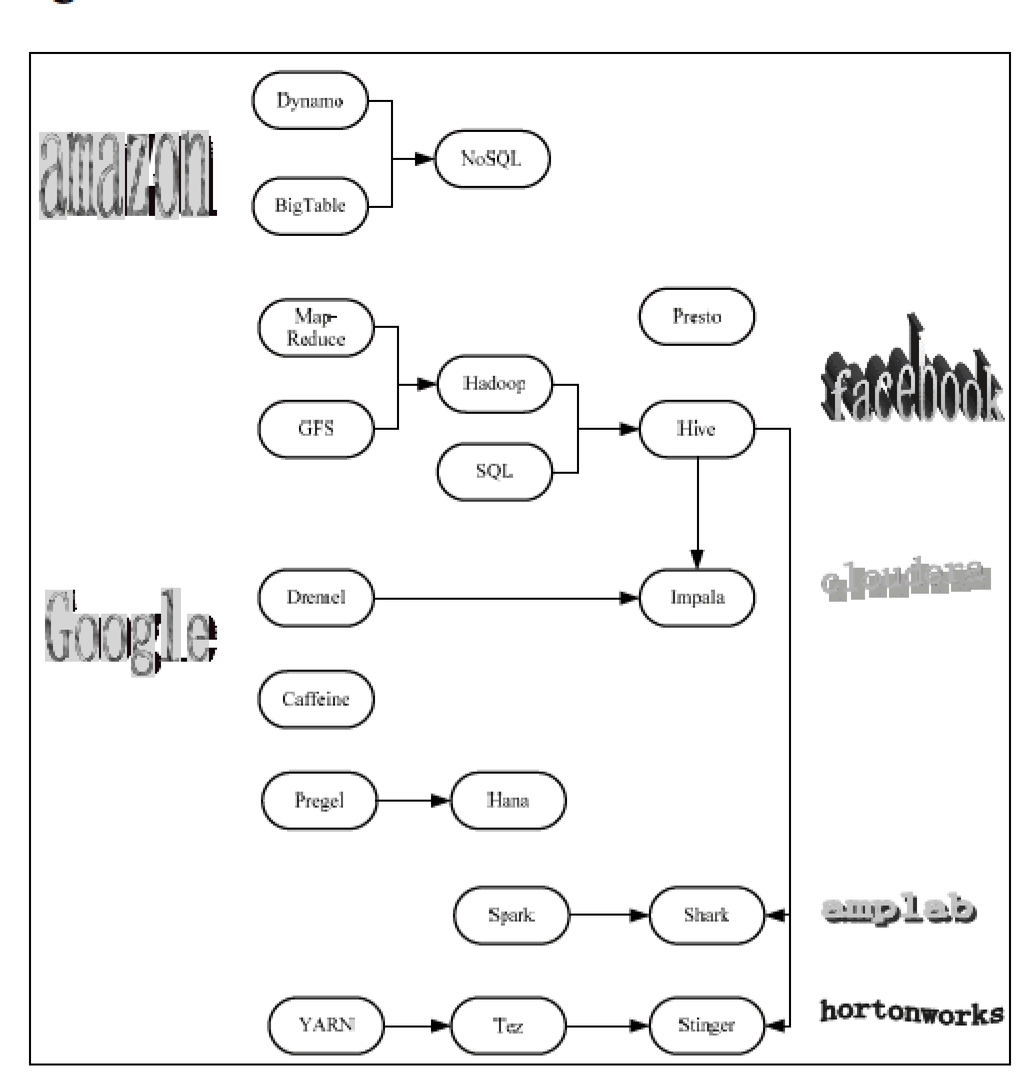


图 1-1　Hive、Impala、Shark、Stinger 和 Presto 的进化图谱

基于 Map-Reduce 模式的 Hadoop 擅长数据批处理，不是特别符合即时查询的场景。实时查询一般使用 MPP (Massively Parallel Processing)的架构，因此用户需要在 Hadoop 和 MPP 两种技术中选择。在 Google 的第二波技术浪潮中，一些基于 Hadoop 架构的快速 SQL 访问技术逐步获得人们关注。现在有一种新的趋势是 MPP 和 Hadoop 相结合提供快速 SQL 访问框架。最近有 4 个很热门的开源工具出来：Impala、Shark、Stinger 和 Presto，这也显示了大数据领域对于 Hadoop 生态系统中支持实时查询的期望。总体来说，Impala、Shark、Stinger 和 Presto 4 个系统都是类 SQL 实时大数据查询分析引擎，但是它们的技术侧重点完全不同，而且它们也不是为了替换 Hive 而生，Hive 在做数据仓库时是非常有价值的。这 4 个系统与 Hive 都是构建在 Hadoop 之上的数据查询工具，各有不同的侧重适应面，但从客户端使用来看它们与 Hive 有很多的共同之处，如数据表元数据、Thrift 接口、ODBC/JDBC 驱动、SQL 语法、灵活的文件格式、存储资源池等。Hive 与 Impala、Shark、Stinger、Presto 在 Hadoop 中的关系如图 1-2 所示。Hive 适用于长时间的批处理查询分析，而 Impala、Shark、Stinger 和 Presto 适用于实时交互式 SQL 查询，它们给数据分析人员提供了快速实验、验证想法的大数据分析工具。可以先使用 Hive 进行数据转换处理，之后使用这 4 个系统中的一个在 Hive 处理后的结果数据集上进行快速的数据分析。

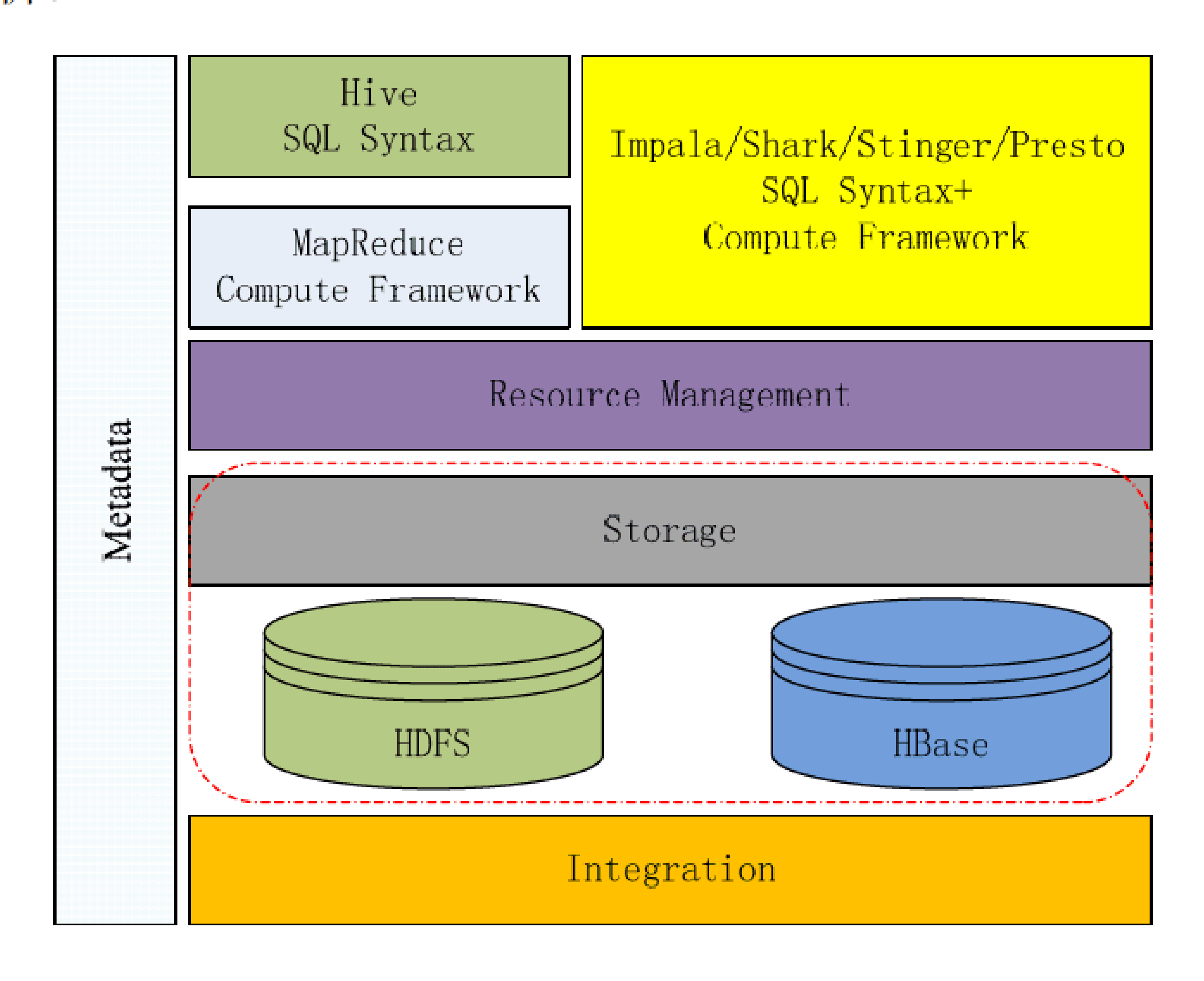


图 1-2　Hive 与 Impala、Shark、Stinger、Presto 在 Hadoop 中的关系

2. 批处理计算 MapReduce、Spark 等

最适合于完成大数据批处理的计算模式是 MapReduce，这是 MapReduce 设计之初的主要任务和目标。MapReduce 是一个单输入、两阶段（Map 和 Reduce）的数据处理过程。首先，MapReduce 对具有简单数据关系、易于划分的大规模数据采用“分而治之”的并行处理思想；然后将大量重复的数据记录处理过程总结成 Map 和 Reduce 两个抽象的操作；最后 MapReduce 提供了一个统一的并行计算框架，把并行计算所涉及的诸多系统层细节都交给计算框架去完成，以此大大简化了程序员进行并行化程序设计的负担。

MapReduce 的简单易用性使其成为目前大数据处理最成功的主流并行计算模式。在开源社区的努力下，开源的 Hadoop 系统目前已成为较为成熟的大数据处理平台，并已发展成一个包括众多数据处理工具和环境的完整的生态系统。目前几乎国内外的各个著名 IT 企业都在使用 Hadoop 平台进行企业内大数据的计算处理。此外，Spark 系统也具备批处理计算的能力。

3. 流式计算 Scribe、Flume、Storm、S4、Spark Steaming 等

流式计算无法确定数据的到来时刻和到来顺序，也无法将全部数据存储起来。因此，不再进行流式数据的存储，而是当流动的数据到来后在内存中直接进行数据的实时计算。如 Twitter 的 Storm、Yahoo 的 S4 就是典型的流式数据计算架构，数据在任务拓扑中被计算，并输出有价值的信息。流式计算和批量计算分别适用于不同的大数据应用场景：对于先存储后计算，实时性要求不高，同时，数据的准确性、全面性是更为重要的应用场景，批量计算模式更合适；对于无须先存储，可以直接进行数据计算，实时性要求很严格，但数据的精确度要求稍微宽松的应用场景，流式计算具有明显优势。流式计算中，数据往往是最近一个时间窗口内的，因此数据延迟往往较短，实时性较强，但数据的精确程度往往较低。流式计算和批量计算具有明显的优劣互补特征，在多种应用场合下可以将两者结合起来使用。通过发挥流式计算的实时性优势和批量计算的计算精度优势，满足多种应用场景在不同阶段的数据计算要求。

4. 迭代计算 HaLoop、iMapReduce、Twister、Spark 等

传统的 MapReduce 框架把一个作业的执行过程分为两个阶段：map 和 reduce，在 map 阶段，每个 map task 读取一个 block，并调用 map()函数进行处理，然后将结果写到本地磁盘上；在 reduce 阶段，每个 reduce task 远程地从 map task 所在节点上读取数据，调用 reduce()函数进行数据处理，并将最终结果写到 HDFS。从以上过程可以看出，map 阶段和 reduce 阶段的结果均要写磁盘，这虽然会降低系统性能，但可以提高可靠性。正是由于这个原因，传统的 MapReduce 不能显式地支持迭代编程，如果用户硬要在传统 MapReduce 上运行迭代式作业，性能将非常低。为了克服 Hadoop MapReduce 难以支持迭代计算的缺陷，工业界和学术界对 Hadoop MapReduce 进行了改进研究，不少改进型的 MapReduce 出现了，它们能很好地支持迭代式开发。目前，一个具有快速和灵活的迭代计算能力的典型系统是 Spark，其采用了基于内存的 RDD 数据集模型实现快速的迭代计算。

5. 图计算 Pregel、Giraph、Trinity、PowerGraph、GraphX 等

目前已经出现了很多分布式图计算系统，其中较为典型的系统包括 Google 公司的 Pregel、Facebook 对 Pregel 的开源实现 Giraph、微软公司的 Trinity、Spark 下的 GraphX、CMU 的 GraphLab 以及由其衍生出来的目前性能最快的图数据处理系统 PowerGraph。

6. 内存计算 Dremel、Hana、Redis 等

随着内存价格的不断下降以及服务器可配置的内存容量的不断提高，用内存计算完成高速的大数据处理已经成为大数据计算的一个重要发展趋势。例如，Hana 系统设计者总结了很多实际的商业应用后发现，一个提供 50TB 总内存容量的计算集群将能够满足绝大多数现有的商业系统对大数据的查询分析处理要求，如果一个服务器节点可配置 1TB～2TB 的内存，则需要 25～50 个服务器节点。目前 Intel Xeon E-7 系列处理器最大可支持高达 1.5TB 的内存，因此，配置一个上述大小规模的内存计算集群是可以做到的。

1.2 数据挖掘及其相关领域应用

大数据时代，虽然数据安全被一而再地强调，但是人们显然更乐于大数据和数据发掘的探索。从巨量数据中提取出有用的信息，创造有用的价值都是各个领域在不断努力的方向。

1.2.1 数据挖掘概述

数据挖掘（Data mining），又译为资料探勘、数据采矿，它是数据库知识发现（Knowledge-Discovery in Databases) KDD 中的一个步骤。数据挖掘一般是指从大量的数据中通过算法搜索隐藏于其中信息的过程。数据挖掘通常与计算机科学有关，并通过统计、在线分析处理、情报检索、机器学习、专家系统（依靠过去的经验法则）和模式识别等诸多方法来实现上述目标。

数据挖掘是一个从数据中提取模式的过程，是一个受多个学科影响的交叉领域，包括数据库系统、统计学、机器学习、可视化和信息科学等。数据挖掘反复使用多种数据挖掘算法从观测数据中确定模式或合理模型，是一种决策支持过程，通过预测客户的行为，帮助企业的决策者调整市场策略，减少风险，做出正确的决策。由于传统的事物型工具（如查询工具、报表工具）无法回答事先未定义的综合性问题或跨部门/机构的问题，因此其用户必须清楚地了解问题的目的。数据挖掘就可以回答事先未加定义的综合性问题或跨部门/机构的问题，挖掘潜在的模式并预测未来的趋势，用户不必提出确切的问题，而且模糊问题更有利于发现未知的事实。图 1-3 为数据挖掘过程。图 1-4 为数据挖掘系统结构。

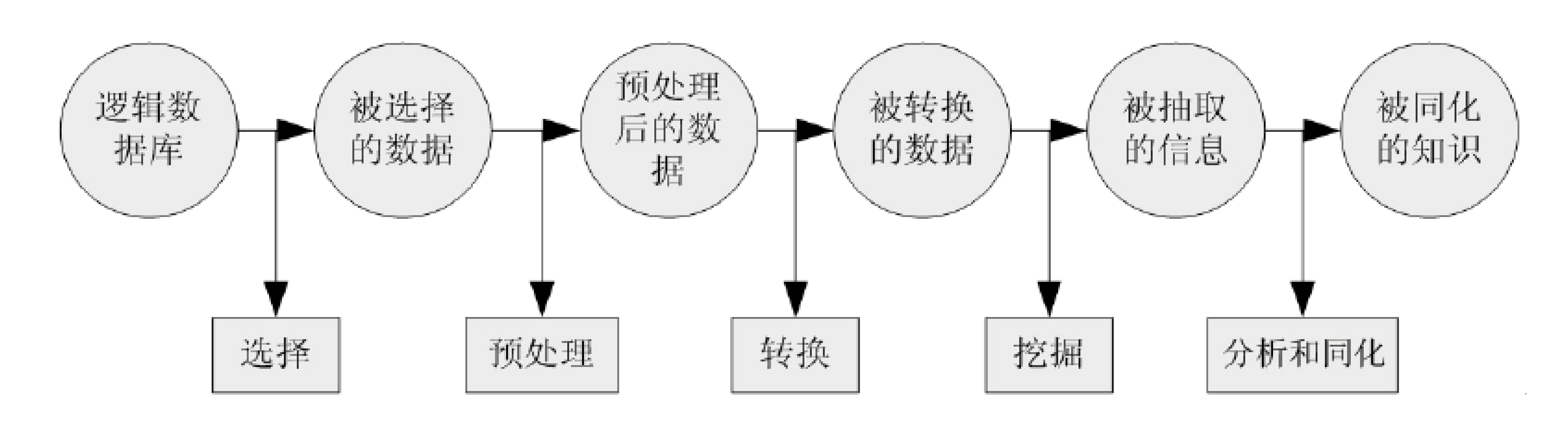


图 1-3　数据挖掘过程

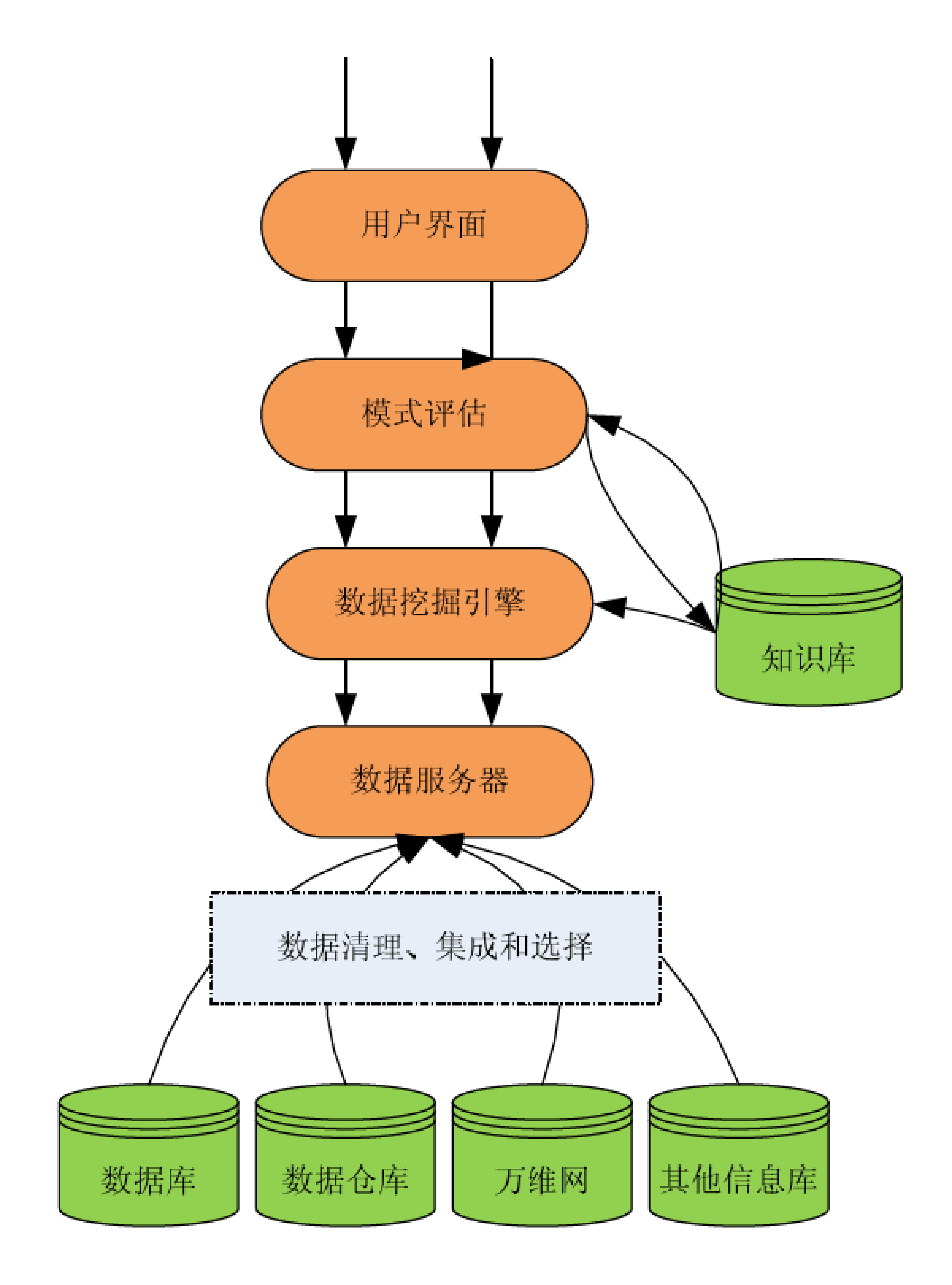


图 1-4　数据挖掘系统结构

数据挖掘有很多种分类方法，如可按发现的知识种类、挖掘的数据库类型、挖掘方法、挖掘途径、所采用的技术等分类。下面只讨论 4 个应用比较广泛的方法。

1. 关联规则（Association Rule）

在数据挖掘领域中，关联规则应用最为广泛，是重要的研究方向，表示数据库中一组对象之间某种关联关系的规则。一般来讲，可以用多个参数来描述一个关联规则的属性，常用的有：可信度、支持度、兴趣度、期望可信度、作用度。

2. 离群数据（Outlier）

离群数据就是明显偏离其他数据、不满足数据的一般模式或行为、与存在的其他数据不一致的数据。数据挖掘的大部分研究忽视了离群数据的存在和意义，现有的方法往往研究如何减少离群数据对正常数据的影响，或仅仅把其当作噪音来对待。这些离群数据可能来源于计算机录入错误、人为错误等，也可能就是数据的真实反映。

3. 基于案例的推理（case-based reasoning, CBR）

基于案例的推理来源于人类的认知心理活动，它属于类比推理方法。其基本思想是基于人们在问题求解中习惯于过去处理类似问题的经验和获取的知识，再针对新旧情况的差异做相应的调整，从而得到新问题的解并形成新的案例。CBR 方法的应用越来越受到人们的重

视，在许多领域都有较好的推广前景，例如，在气象、环保、地震、农业、医疗、商业、CAD 等领域；CBR 也可用在计算机软硬件的生产中，如软件及硬件的故障检测；CBR 方法尤其在不易总结出专家知识的领域中，应用越来越普遍，也越来越深入。

4. 支持向量机（Support Vector Machine，SVM）

支持向量机是近几年发展起来的新型通用的知识发现方法，在分类方面具有良好的性能。SVM 是建立在计算学习理论结构风险的最小化原则之上，主要思想是针对两类分类问题在高位空间中寻找一个超平面作为两类的分割，以保证最小的分类错误率。

1.2.2　数据挖掘与机器学习

数据挖掘受到很多学科领域的影响，其中数据库、机器学习、统计学无疑影响最大。简言之，对数据挖掘而言，数据库提供数据管理技术，机器学习和统计学提供数据分析技术。由于统计学往往醉心于理论的优美而忽视实际的效用，因此，统计学界提供的很多技术通常都要在机器学习界进一步研究，变成有效的机器学习算法之后才能再进入数据挖掘领域。从这个意义上说，统计学主要是通过机器学习来对数据挖掘发挥影响，而机器学习和数据库则是数据挖掘的两大支撑技术。从数据分析的角度来看，绝大多数数据挖掘技术都来自机器学习领域，但机器学习研究往往并不把海量数据作为处理对象，因此，数据挖掘要对算法进行改造，使得算法性能和空间占用达到实用的地步。

“机器学习”是人工智能的核心研究领域之一，其最初的研究动机是为了让计算机系统具有人的学习能力以便实现人工智能，因为众所周知，没有学习能力的系统很难被认为是具有智能的。目前被广泛采用的机器学习的定义是“利用经验来改善计算机系统自身的性能”。事实上，由于“经验”在计算机系统中主要是以数据的形式存在的，因此机器学习需要设法对数据进行分析，这就使得它逐渐成为智能数据分析技术的创新源之一，并且为此而受到越来越多的关注。

“数据挖掘”和“知识发现”通常被相提并论，并在许多场合被认为是可以相互替代的术语。对数据挖掘有多种文字不同但含义接近的定义，例如“识别出巨量数据中有效的、新颖的、潜在有用的、最终可理解的模式的非平凡过程”。其实顾名思义，数据挖掘就是试图从海量数据中找出有用的知识。大体上看，数据挖掘可以视为机器学习和数据库的交叉，它主要利用机器学习界提供的技术来分析海量数据，利用数据库界提供的技术来管理海量数据。

1.2.3　数据挖掘与数据库

数据挖掘利用了来自如下一些领域的思想：

（1）来自统计学的抽样、估计和假设检验。

（2）人工智能、模式识别和机器学习的搜索算法、建模技术和学习理论。

数据挖掘也迅速地接纳了来自其他领域的思想，这些领域包括最优化、进化计算、信息论、信号处理、可视化和信息检索。一些其他领域也起到重要的支撑作用，特别地，需要数

据库系统提供有效的存储、索引和查询处理支持。

要将庞大的数据转换成为有用的信息，必须先有效率地收集信息。随着科技的进步，功能完善的数据库系统就成了最好的收集数据的工具。数据仓库，简单地说，就是搜集来自其他系统的有用数据，存放在一整合的储存区内。所以其实就是一个经过处理整合，且容量特别大的关系型数据库，用以储存决策支持系统（Decision Support System）所需的数据，供决策支持或数据分析使用。从信息技术的角度来看，数据仓库的目标是在组织中，在正确的时间，将正确的数据交给正确的人。许多人对于 Data Warehousing 和 Data Mining 时常混淆，不知如何分辨。其实，数据仓库是数据库技术的一个新主题，利用计算机系统帮助我们操作、计算和思考，让作业方式改变，决策方式也跟着改变。

1.2.4 数据挖掘与统计学

数据挖掘源自于统计分析，而又不同于统计分析。数据挖掘不是为了替代传统的统计分析技术，相反，数据挖掘是统计分析方法的扩展和延伸。大多数的统计分析技术都基于完善的数学理论和高超的技巧，其预测的准确程度还是令人满意的，但对于使用者的知识要求比较高。而随着计算机能力的不断发展，数据挖掘可以利用相对简单和固定程序完成同样的功能。新的计算算法的产生如神经网络、决策树使人们不需了解到其内部复杂的原理也可以通过这些方法获得良好的分析和预测效果。

由于数据挖掘和统计分析根深蒂固的联系，通常的数据挖掘工具都能够通过可选件或自身提供统计分析功能。这些功能对于数据挖掘的前期数据探索和数据挖掘之后对数据进行总结和分析都是十分必要的。统计分析所提供的诸如方差分析、假设检验、相关性分析、线性预测、时间序列分析等功能都有助于数据挖掘前期对数据进行探索：发现数据挖掘的题目、找出数据挖掘的目标、确定数据挖掘所需涉及的变量、对数据源进行抽样等。所有这些前期工作对数据挖掘的效果产生重大影响，而数据挖掘的结果也需要统计分析的描述功能（最大值、最小值、平均值、方差、四分位、个数、概率分配）进行具体描述，使数据挖掘的结果能够被用户了解。因此，统计分析和数据挖掘是相辅相成的过程，两者的合理配合是数据挖掘成功的重要条件。

1.2.5 数据挖掘与决策支持

数据挖掘的知识通常表现为概念、规则、规律、模式、约束和可视化等形式。这些知识经过解释后可以直接在实际系统中应用，以辅助决策过程；或者提供给领域专家，修正专家已有的知识体系；也可以作为新的知识转存到应用系统的知识库中。发现的过程是使数据挖掘利用各种知识发现算法，从数据库中发现、表达、更新和解释有关知识的过程。

作为信息处理新发展阶段的决策支持系统尚处于初级阶段，投入应用的成功实例并不多。有些开发的系统仅为简单的查询系统或报表系统，并不能给决策者提供辅助决策信息。分析原因主要有以下几个方面：

（1）决策支持系统需要以集成数据为基础，然而现实中的数据往往是分散管理的且大多分布于异构的数据平台，数据集成不易。

（2）决策支持涉及大量历史数据和半结构化问题，传统的数据库管理系统因自身的局限

性并不提供这些方面的支持。

（3）决策支持系统的建立需要对数据、模型、知识和接口进行集成。数据库语言数值计算能力较低，因而采用数据库管理技术建立决策支持系统在辅助决策支持方面知识表达、知识综合和知识推理能力比较薄弱，难以满足人们日益提高的决策要求。近年来，数据挖掘技术的发展给以上问题的解决带来了新的契机。

1.2.6　数据挖掘与云计算

人类社会信息正以“每 18 个月产生的数量等于过去几千年的总和”的速度不断增加，如此浩瀚的数据在带给人们大量信息的同时，也极大地增加了人们从海量数据中发现有用知识的难度，而解决这一问题的努力促进了云计算和数据挖掘技术的结合和快速发展。

按照中国电子学会云计算专家委员会的技术白皮书阐述，云计算是一种基于互联网的、大众参与的计算模式，其计算资源（计算能力、存储能力、交互能力）是动态、可伸缩，且被虚拟化的，而且以服务的方式提供。

数据挖掘远比信息搜索要复杂。过去对海量数据的处理主要是通过高性能机或者更大规模的计算设备来实现，现在通过基于云计算的数据挖掘能更好地达到目的。采用云计算模式有许多好处，成本低廉、容错性强、计算速度快、程序开发便捷、节点的增加更容易。可以说云计算是数据挖掘中普遍适用，较为理想的计算模式，也是我们从海量数据中找到有用、可理解的知识的技术手段。

通过云计算的海量数据存储和分布计算，为云计算环境下的海量数据挖掘提供了新的方法和手段，有效解决了海量数据挖掘的分布存储和高效计算问题。开展基于云计算的数据挖掘方法的研究，可以为更多、更复杂的海量数据挖掘提供新的理论与支撑工具。而作为传统数据挖掘向云计算的延伸将推动互联网技术成果服务于大众，是促进信息资源的深度分享和可持续利用的新方法、新途径。

1.3　大数据应用

不计其数的互联网用户和机器间的连接导致数据呈爆炸式增长。使用大数据需要将信息基础设施转型为一个更加灵活、更具分布式且更开放的环境。了解什么是大数据和大数据对行业的意义之后，即可开始着手规划大数据项目。确定合适的时机，评估当前的环境，分析项目需求并确定技术需要。

1.3.1　大数据应用案例

什么是大数据？我们不再举例说啤酒和尿布的例子了，Gartner 的分析师 Doug Laney 在讲解大数据案例时提到过 8 个更有新意、更典型的案例，可帮助我们更清晰地理解大数据时代的到来。

（1）梅西百货的实时定价机制。根据需求和库存的情况，该公司基于 SAS 的系统对多

达 7300 万种货品进行实时调价。

（2）Tipp24 AG 针对欧洲博彩业构建的下注和预测平台。该公司用 KXEN 软件来分析数十亿计的交易以及客户的特性，然后通过预测模型对特定用户进行动态的营销活动。这项举措减少了 90%的预测模型构建时间。SAP 公司正在试图收购 KXEN。

（3）沃尔玛的搜索。这家零售业寡头为其网站 Walmart.com 自行设计了最新的搜索引擎 Polaris，利用语义数据进行文本分析、机器学习和同义词挖掘等。根据沃尔玛的说法，语义搜索技术的运用使得在线购物的完成率提升了 10%到 15%。“对沃尔玛来说，这就意味着数十亿美元的金额。”Laney 说。

（4）快餐业的视频分析。该公司通过视频分析等候队列的长度，然后自动变化电子菜单显示的内容。如果队列较长，则显示可以快速供给的食物；如果队列较短，则显示那些利润较高但准备时间相对长的食品。

（5）Morton 牛排店的品牌认知。当一位顾客开玩笑地通过推特向这家位于芝加哥的牛排连锁店订餐送到纽约 Newark 机场（他将在一天工作之后抵达该处）时，Morton 就开始了自己的社交秀。首先，分析推特数据，发现该顾客是本店的常客，也是推特的常用者。根据客户以往的订单，推测出其所乘的航班，然后派出一位身着燕尾服的侍者为客户提供晚餐。

（6）PredPol Inc.。PredPol 公司通过与洛杉矶和圣克鲁斯的警方以及一群研究人员合作，基于地震预测算法的变体和犯罪数据来预测犯罪发生的几率，可以精确到 500 平方英尺的范围内。在洛杉矶运用该算法的地区，盗窃罪和暴力犯罪分布下降了 33%和 21%。

（7）Tesco PLC（特易购）和运营效率。这家超市连锁在其数据仓库中收集了 700 万部冰箱的数据。通过对这些数据的分析，进行更全面的监控并进行主动的维修以降低整体能耗。

（8）American Express（AmEx，美国运通）和商业智能。以往，AmEx 只能实现事后诸葛式的报告和滞后的预测。“传统的 BI 已经无法满足业务发展的需要。”Laney 认为。于是，AmEx 开始构建真正能够预测忠诚度的模型，基于历史交易数据，用 115 个变量来进行分析预测。该公司表示，对于澳大利亚将于之后四个月中流失的客户，已经能够识别出其中的 24%。

1.3.2 大数据应用场景

当我们最初接触大数据的时候，谈得最多的可能是用户行为分析，即通过各种用户行为，包括浏览记录、消费记录、交往和购物娱乐、行动轨迹等产生的数据。由于这些数据本身符合海量、异构的特征，同时通过分析这些数据之间的关联性容易匹配某些结果现象。即有一堆的行为因子 x，同时又有一堆的结果构成 y，我们找寻到了某种相关性，有利于我们调整后续的各种策略。

为何 Google 能够做大数据？你思考过吗？因为搜索本身往往是用户行为的一个重要入口，即搜索引擎具备了实时采集多个用户行为的 x 因子的能力。而这个能力往往是单个电商门户网站无法做到的。但是搜索引擎做大数据的弱势在哪里？即前面谈到的用户和用户之间的关系较难建立，而更多的是本身行为之间的相关性。从这个差异上也可以看到搜索引擎更加容易做交通、疾病、气象等方面的大数据分析和预测；而类似电商平台或类似腾讯更加容

易做消费和娱乐类的大数据分析和预测。

对于大数据的应用场景，包括各行各业对大数据处理和分析的应用，最核心的还是用户需求。接下来，本文通过梳理各个行业在大数据应用领域面临的挑战、如何寻找突破口来展示其潜在存在的大数据应用场景。

1. 医疗大数据，医患看病更高效

除了较早前就开始利用大数据的互联网公司，医疗行业是让大数据分析最先发扬光大的传统行业之一。医疗行业拥有大量的病例、病理报告、治愈方案、药物报告等。如果这些数据可以被整理和应用将会极大地帮助医生和病人。我们面对的数目及种类众多的病菌、病毒，以及肿瘤细胞，其都处于不断地进化的过程中。在发现诊断疾病时，疾病的确诊和治疗方案的确定是最困难的。

在未来，借助于大数据平台我们可以收集不同病例和治疗方案，以及病人的基本特征，可以建立针对疾病特点的数据库。如果未来基因技术发展成熟，可以根据病人的基因序列特点进行分类，建立医疗行业的病人分类数据库。在医生诊断病人时可以参考病人的疾病特征、化验报告和检测报告，参考疾病数据库来快速帮助病人确诊，明确定位疾病。在制定治疗方案时，医生可以依据病人的基因特点，调取相似基因、年龄、人种、身体情况相同的有效治疗方案，制定出适合病人的治疗方案，帮助更多人及时进行治疗。同时这些数据也有利于医疗行业开发出更加有效的药物和医疗器械。

医疗行业的数据应用一直在进行，但是数据没有打通，都是孤岛数据，没有办法进行大规模应用。未来需要将这些数据统一收集起来，纳入统一的大数据平台，为人类健康造福。政府和医疗行业是推动这一趋势的重要动力。

2. 生物大数据，人类改良基因策略基础

自人类基因组计划完成以来，以美国为代表，世界主要发达国家纷纷启动了生命科学基础研究计划，如国际千人基因组计划、DNA 百科全书计划、英国十万人基因组计划等。这些计划引领生物数据呈爆炸式增长，目前每年全球产生的生物数据总量已达 EB 级，生命科学领域正在爆发一次数据革命，生命科学某种程度上已经成为大数据科学。

我们来看看今天的准妈妈们，除了要准备尿布、奶瓶和婴儿装，她们还会把基因测试列入计划单。基因测试能让未来的父母对于他们未出生的 baby 的健康有更多的了解。对基因携带者筛查和胚胎植入前诊断，使一个家庭孕育小孩的过程产生了巨大改变。

当下，我们所说的生物大数据技术主要是指大数据技术在基因分析上的应用，通过大数据平台人类可以将自身和生物体基因分析的结果进行记录和存储，利用建立基于大数据技术的基因数据库，大数据技术将会加速基因技术的研究，快速帮助科学家进行模型的建立和基因组合模拟计算。基因技术是人类未来战胜疾病的重要武器，借助于大数据技术的应用，人们将会加快自身基因和其他生物的基因的研究进程。未来利用生物基因技术来改良农作物、利用基因技术来培养人类器官、利用基因技术来消灭害虫都即将实现。

与全球蒸蒸日上的生物大数据创新发展热潮相比，中国的研发及应用才拉开帷幕。我国有四大方面非常欠缺：其一，国内现有的生物大数据分析能力虽然与欧美相差不大，但是在数据分析构架、软件系统与先进的 IT 技术接轨上有待提升；其二，国外在生物大数据领域的

领先人才多，尽管我们也有国际顶级刊物上发表的论文和成果，总体而言，国内高水准团队还是少；其三，欧美讲求成果应用，层出不穷的分析软件可被实验室、临床、产业多方应用；其四，在生物大数据理论研究、标准制定和广泛应用上，中国都亟待全面跟进。

3. 金融大数据，金融行业理财利器

金融行业的大数据面临的往往是同样的问题，但是情况可能要好点：类似企业和个人的一些信用记录现在有全国性质的统一数据库能够拿到部分数据，但是对于单个银行来说，同样是无法拿到用户在其他银行的行为记录数据的；其二银行本身在做很多信贷风险分析的时候，确实需要大量数据做相关性分析，但是很多数据来源于政府各个职能部门，包括工商税务、质量监督、检察院法院等，这些数据短期仍然是无法拿到；还有就是企业或个人从事日常产生的各种行为数据更难拿到；那么，对客户的风险性评估还是得借用原来的老方法。

大数据在金融行业应用范围较广，典型的案例有花旗银行利用 IBM 沃森电脑为财富管理客户推荐产品；美国银行利用客户点击数据集为客户提供特色服务，如有竞争的信用额度；招商银行利用客户刷卡、存取款、电子银行转账、微信评论等行为数据进行分析，每周给客户发送针对性广告信息，里面有顾客可能感兴趣的产品和优惠信息。

可见，大数据在金融行业的应用可以总结为以下 5 个方面：

- 精准营销：依据客户消费习惯、地理位置、消费时间进行推荐。
- 风险管控：依据客户消费和现金流提供信用评级或融资支持，利用客户社交行为记录实施信用卡反欺诈。
- 决策支持：利用决策树技术抵押贷款管理，利用数据分析报告实施产业信贷风险控制。
- 效率提升：利用金融行业全局数据了解业务运营薄弱点，利用大数据技术加快内部数据处理速度。
- 产品设计：利用大数据计算技术为财富客户推荐产品，利用客户行为数据设计满足客户需求的金融产品。

4. 零售大数据，让企业最懂消费者

零售行业大数据应用有两个层面：一个层面是零售行业可以了解客户消费喜好和趋势，进行商品的精准营销，降低营销成本。另一层面是依据客户购买产品，为客户提供可能购买的其他产品，扩大销售额，也属于精准营销范畴。另外，零售行业可以通过大数据掌握未来消费趋势，有利于热销商品的进货管理和过季商品的处理。零售行业的数据对于产品生产厂家是非常宝贵的，零售商的数据信息将会有助于资源的有效利用，降低产能过剩，厂商依据零售商的信息按实际需求进行生产，减少不必要的生产浪费。

未来考验零售企业的不再只是零供关系的好坏，而是要看挖掘消费者需求，以及高效整合供应链满足其需求的能力，因此信息科技技术水平的高低成为获得竞争优势的关键要素。不论是国际零售巨头，还是本土零售品牌，要想顶住日渐微薄的利润率带来的压力，在这片红海中立于不败之地，就必须思考如何拥抱新科技，并为顾客们带来更好的消费体验。

想象一下这样的场景，当顾客在地铁候车时，墙上有某一零售商的巨幅数字屏幕广告，

可以自由浏览产品信息，对感兴趣的或需要购买的商品用手机扫描下单，约定在晚些时候送到家中。而在顾客浏览商品并最终选购商品后，商家已经了解顾客的喜好及个人详细信息，按要求配货并送达顾客家中。未来，甚至顾客都不需要有任何购买动作，利用之前购买行为产生的大数据，当你的沐浴露剩下最后一滴时，你中意的沐浴露就已送到你的手上，而虽然顾客和商家从未谋面，但已如朋友般熟识。

5. 电商大数据，实现精准营销法宝

电商是最早利用大数据进行精准营销的行业，除了精准营销，电商可以依据客户消费习惯来提前为客户备货，并利用便利店作为货物中转点，在客户下单 15 分钟内将货物送上门，提高客户体验。马云的菜鸟网络宣称的 24 小时完成在中国境内的送货，以及京东刘强东宣传未来京东将在 15 分钟完成送货上门都是基于客户消费习惯的大数据分析和预测。

电商可以利用其交易数据和现金流数据，为其生态圈内的商户提供基于现金流的小额贷款，电商业也可以将此数据提供给银行，同银行合作为中小企业提供信贷支持。由于电商的数据较为集中，数据量足够大，数据种类较多，因此未来电商数据应用将会有更多的想象空间，包括预测流行趋势、消费趋势、地域消费特点、客户消费习惯、各种消费行为的相关度、消费热点、影响消费的重要因素等。依托大数据分析，电商的消费报告将有利于品牌公司产品设计、生产企业的库存管理和计划生产、物流企业的资源配置、生产资料提供方产能安排等，有利于精细化社会化大生产，有利于精细化社会的出现。

6. 农牧大数据，提供量化生产参考

大数据在农业中的应用主要是指依据未来商业需求的预测来进行农牧产品生产，降低菜贱伤农的概率。同时大数据的分析将会更加精确地预测未来的天气气候，帮助农牧民做好自然灾害的预防工作。大数据同时也会帮助农民依据消费者消费习惯来决定增加哪些品种的种植，减少哪些品种农作物的生产，提高单位种植面积的产值，有助于快速销售农产品，完成资金回流。牧民可以通过大数据分析来安排放牧范围，有效利用牧场。

由于农产品不容易保存，因此合理种植和养殖农产品十分重要。如果没有规划好，容易产生菜贱伤农的悲剧。过去出现的猪肉过剩、卷心菜过剩、香蕉过剩的原因就是农牧业没有规划好。借助于大数据提供的消费趋势报告和消费习惯报告，政府将为农牧业生产提供合理引导，建议依据需求进行生产，避免产能过剩，造成不必要的资源和社会财富浪费。农业关乎国计民生，科学的规划将有助于社会整体效率提升。大数据技术可以帮助政府实现农业的精细化管理，实现科学决策。在数据驱动下，结合无人机技术，农民可以采集农产品生长信息，病虫害信息。相对于过去雇佣飞机成本将大大降低，同时精度也将大大提高。

7. 交通大数据，智慧交通畅通出行

交通作为人类行为的重要组成和重要条件之一，对于大数据的感知也是最急迫的。近年来，我国的智能交通已实现了快速发展，许多技术手段都达到了国际领先水平。但是，问题和困境也非常突出，从各个城市的发展状况来看，智能交通的潜在价值还没有得到有效挖掘：对交通信息的感知和收集有限，对存在于各个管理系统中的海量的数据无法共享运用、有效分析，对交通态势的研判预测乏力，对公众的交通信息服务很难满足需求。这虽然有各

地在建设理念、投入上的差异，但是整体上智能交通的现状是效率不高，智能化程度不够，使得很多先进技术设备发挥不了应有的作用，也造成了大量投入上的资金浪费。这其中很重要的问题是小数据时代带来的硬伤：从模拟时代带来的管理思想和技术设备只能进行一定范围的分析，而管理系统的那些关系型数据库只能刻板地分析特定的关系，对于海量数据尤其是半结构、非结构数据无能为力。

尽管现在已经基本实现了数字化，但是数字化和数据化还根本不是一回事，只是局部地提高了采集、存储和应用的效率，本质上并没有太大的改变。而大数据时代的到来必然带来破解难题的重大机遇。大数据必然要求我们改变小数据条件下一味地精确计算，而是更好地面对混杂，把握宏观态势；大数据必然要求我们不再热衷因果关系而是相关关系，使得处理海量非结构化数据成为可能；也必然促使我们努力把一切事物数据化，最终实现管理的便捷高效。

目前，交通的大数据应用主要在两个方面：一方面可以利用大数据传感器数据来了解车辆通行密度，合理进行道路规划，包括单行线路规划；另一方面可以利用大活数据来实现即时信号灯调度，提高已有线路运行能力。科学地安排信号灯是一个复杂的系统工程，必须利用大数据计算平台才能计算出一个较为合理的方案。科学的信号灯安排将会提高 30%左右已有道路的通行能力。在美国，政府依据某一路段的交通事故信息来增设信号灯，降低了 50%以上的交通事故率。机场的航班起降依靠大数据将会提高航班管理的效率，航空公司利用大数据可以提高上座率，降低运行成本。铁路利用大数据可以有效安排客运和货运列车，提高效率、降低成本。

8. 教育大数据，建立因材施教系统工程

随着技术的发展，信息技术已在教育领域有了越来越广泛的应用。考试、课堂、师生互动、校园设备使用、家校关系等只要技术达到的地方，各个环节都被数据包裹。

在课堂上，数据不仅可以帮助改善教育教学，在重大教育决策制定和教育改革方面，大数据更有用武之地。美国利用数据来诊断处在辍学危险期的学生、探索教育开支与学生学习成绩提升的关系、探索学生缺课与成绩的关系。举一个比较有趣的例子，教师的高考成绩和所教学生的成绩有关吗？究竟如何，不妨借助数据来看。比如美国某州公立中小学的数据分析显示，在语文成绩上，教师高考分数和学生成绩呈现显著的正相关。也就是说，教师的高考成绩与他们现在所教语文课上的学生学习成绩有很明显的关系，教师的高考成绩越好，学生的语文成绩也越好。这个关系让我们进一步探讨其背后真正的原因。其实，教师高考成绩高低某种程度上是教师的某个特点在起作用，而正是这个特点对教好学生起着至关重要的作用，教师的高考分数可以作为挑选教师的一个指标。如果有了充分的数据，便可以发掘更多的教师特征和学生成绩之间的关系，从而为挑选教师提供更好的参考。

大数据还可以帮助家长和教师甄别出孩子的学习差距和有效的学习方法。比如，美国的麦格劳-希尔教育出版集团就开发出了一种预测评估工具，帮助学生评估他们已有的知识和达标测验所需程度的差距，进而指出学生有待提高的地方。评估工具可以让教师跟踪学生学习情况，从而找到学生的学习特点和方法。有些学生适合按部就班，有些则更适合图式信息和整合信息的非线性学习。这些都可以通过大数据搜集和分析很快识别出来，从而为教育教学提供坚实的依据。

在国内尤其是北京、上海、广东等城市，大数据在教育领域就已有了非常多的应用，譬如像慕课、在线课程、翻转课堂等，其中就应用了大量的大数据工具。

毫无疑问，在不远的将来，无论是针对教育管理部门，还是校长、教师，以及学生和家长，都可以得到针对不同应用的个性化分析报告。通过大数据的分析来优化教育机制，也可以做出更科学的决策，这将带来潜在的教育革命。不久的将来个性化学习终端，将会更多地融入学习资源云平台，根据每个学生的不同兴趣爱好和特长，推送相关领域的前沿技术、资讯、资源乃至未来职业发展方向，等等，并贯穿每个人终身学习的全过程。

9. 体育大数据，提升训练指标创造佳绩

从《点球成金》这部电影开始，体育界的有识之士们终于找到了向往已久的道路，那就是如何利用大数据来让团队发挥最佳水平。从足球到篮球，数据似乎成为赢得比赛甚至是奖杯的金钥匙。

大数据对于体育的改变可以说是方方面面，从运动员本身来讲，可穿戴设备收集的数据让自己更了解身体状况；从媒体评论员来讲，可通过大数据提供的数据更好地解说比赛，分析比赛。数据已经通过大数据分析转化成了洞察力，为体育竞技中的胜利增加筹码，也为身处世界各地的体育爱好者随时随地观赏比赛提供了个性化的体验。

尽管鲜有职业网球选手愿意公开承认自己利用大数据来制定比赛策划和战术，但几乎每一个球员都会在比赛前后使用大数据服务。有教练表示：“在球场上，比赛的输赢取决于比赛策略和战术，以及赛场上连续对打期间的快速反应和决策，但这些细节转瞬即逝，所以数据分析成为一场比赛最关键的部分。对于那些拥护并利用大数据进行决策的选手而言，他们毋庸置疑地将赢得足够竞争优势”。

10. 环保大数据，预警雾霾

2012 年 7 月 21 日北京遭遇特大暴雨，在一天之内，平均降雨量达 164 毫米，也是北京市 61 年以来最大规模暴雨，此次暴雨因来势凶猛给广大市民生活带来巨大影响。其实，摊上这种事儿，最主要的还是需要气象部门及时、准确地做出预警，并协同其他运营商部门，将这种预警信息第一时间下发到北京市民（包括在京旅行的人士）。也正是如此，前年的那场暴雨不仅暴露出了管理工作上的漏洞，也引起了业内人士关于一场“大数据”的探讨。

气象对社会的影响涉及方方面面。传统上依赖气象的主要是农业、林业和水运等行业部门，而如今，气象俨然成为 21 世纪社会发展的资源，并支持定制化服务满足各行各业用户需要。借助于大数据技术，天气预报的准确性和实效性将会大大提高，预报的及时性将会大大提升，同时对于重大自然灾害，例如龙卷风，通过大数据计算平台，人们将会更加精确地了解其运动轨迹和危害的等级，有利于帮助人们提高应对自然灾害的能力。天气预报的准确度的提升和预测周期的延长将会有利于农业生产的安排。

尤其是进入秋冬季以来，我国多个城市爆发雾霾天气，空气污染严重。随着 PM2.5 对于人体健康的危害日益被公众熟知，人们对于“雾霾假”的呼声也越来越高。有人调侃，重度污染天走在上班路上就是一台“人肉吸尘器”。

由此看来，依靠大数据分析北京或其他城市空气污染的形成及对策，任重道远。一是数据的来源。高耗能企业的生产规模、排放量这些数据是否层层上报，准确统计？掌握此数据

的部门是否能向社会公开？北京 500 万辆汽车所加汽油到底有哪些成分，产生的尾气对空气污染指数的“贡献”率到底多大？二是要冲破数据挖掘分析应用的技术壁垒，当然前提就是数据公开。

在美国 NOAA（国家海洋暨大气总署）其实早就在使用大数据业务。每天通过卫星、船只、飞机、浮标、传感器等收集超过 35 亿份观察数据。收集完毕后，NOAA 会汇总大气数据、海洋数据，以及地质数据，进行直接测定，绘制出复杂的高保真预测模型，将其提供给 NWS（国家气象局）做出气象预报的参考数据。目前，NOAA 每年新增管理的数据量就高达 30PB（1PB=1024TB）。由 NWS 生成的最终分析结果，就呈现在日常的天气预报和预警报道上。

11. 食品大数据，舌尖上安全保障

民以食为天，食品安全问题一直是国家的重点关注问题，关系着人们的身体健康和国家安全。近几年，毒胶囊、镉大米、瘦肉精、洋奶粉等食品安全事件不断考验着消费者的承受力，让消费者对食品安全产生了担忧。

近几年外国旅游者减少了到中国旅游，进口食品大幅度增加，这其中一个主要原因就是食品安全问题。随着科学技术和生活水平的不断提高，食品添加剂及食品品种越来越多，传统手段难以满足当前复杂的食品监管需求，从不断出现的食品安全问题来看，食品监管成了食品安全的棘手问题。此刻，通过大数据管理将海量数据聚合在一起，将离散的数据需求聚合能形成数据长尾，从而满足传统中难以实现的需求。在数据驱动下，采集人们在互联网上提供的举报信息，国家可以掌握部分乡村和城市的死角信息，挖出不法加工点，提高执法透明度，降低执法成本。国家可以参考医院提供的就诊信息，分析出涉及食品安全的信息，及时进行监督检查，第一时间进行处理，降低已有不安全食品的危害。参考个体在互联网的搜索信息，掌握流行疾病在某些区域和季节的爆发趋势，及时进行干预，降低其流行危害。政府可以提供不安全食品厂商信息，不安全食品信息，帮助人们提高食品安全意识。

当然，有专业人士认为食品安全涉及从田头到餐桌的每一个环节，需要覆盖全过程的动态监测才能保障食品安全，以稻米生产为例，产地、品种、土壤、水质、病虫害发生、农药种类与数量、化肥、收获、储藏、加工、运输、销售等环节，无一不影响稻米安全状况，通过收集、分析各环节的数据，可以预测某产地将收获的稻谷或生产的稻米是否存在安全隐患。

大数据不仅能带来商业价值，亦能产生社会价值。随着信息技术的发展，食品监管也面临着众多的各种类型的海量数据，如何从中提取有效数据成为关键所在。可见，大数据管理是一项巨大挑战，一方面要及时提取数据以满足食品安全监管需求；另一方面需在数据的潜在价值与个人隐私之间进行平衡。相信大数据管理在食品监管方面的应用，可以为食品安全撑起一把有力的保护伞。

12. 大数据令政府调控和财政支出有条不紊

政府利用大数据技术可以了解各地区的经济发展情况、各产业发展情况、消费支出和产品销售情况，依据数据分析结果，科学地制定宏观政策，平衡各产业发展，避免产能过剩，有效利用自然资源和社会资源，提高社会生产效率。大数据还可以帮助政府进行监控自然资

源的管理，无论是国土资源、水资源、矿产资源、能源等，大数据通过各种传感器来提高其管理的精准度。同时大数据技术也能帮助政府进行支出管理，透明合理的财政支出将有利于提高公信力和监督财政支出。

大数据及大数据技术带给政府的不仅仅是效率提升、科学决策、精细管理，更重要的是数据治国、科学管理的意识改变，未来大数据将会从各个方面来帮助政府实施高效和精细化管理。政府运作效率的提升、决策的科学客观、财政支出合理透明都将大大提升国家整体实力，成为国家竞争优势。大数据带个国家和社会的益处将会具有极大的想象空间。

13. 舆情监控大数据，打破数据边界绘制全景视图

美国密歇根大学研究人员就设计出一种利用“超级计算机以及大量数据”来帮助警方定位那些最易受到不法分子侵扰片区的方法。具体做法是，研究人员通过大量的多类型数据（从人口统计数据到毒品犯罪数据到各区域所出售酒的种类、治安状况、流动人口数据等），创建一张波士顿犯罪高发地区热点图。同时，还将相邻片区等各种因素加入到数据模型中，并根据历史犯罪记录和地点统计不断修正所得出的预测数据。

国家正在将大数据技术用于舆情监控，其收集到的数据除了解民众诉求、降低群体事件之外，还可以用于犯罪管理。大量的社会行为正逐步走向互联网，人们更愿意借助于互联网平台来表述自己的想法和宣泄情绪。社交媒体和朋友圈正成为追踪人们社会行为的平台，正能量的东西有，负能量的东西也不少。一些好心人通过微博来帮助别人寻找走失的亲人或提供可能被拐卖人口的信息，这些都是社会群体互助的例子。国家可以利用社交媒体分享的图片和交流信息，来收集个体情绪信息，预防个体犯罪行为和反社会行为。最近警方通过微博信息抓获了聚众吸毒的人，处罚了虐待小孩的家长。

大数据技术的发展带来企业经营决策模式的转变，驱动着行业变革、衍生出新的商机和发展契机。驾驭大数据的能力已被证实为领军企业的核心竞争力，这种能力能够帮助企业打破数据边界，绘制企业运营全景视图，做出最优的商业决策和发展战略。其实，不论是哪个行业的大数据分析和应用场景，可以看到一个典型的特点还是无法离开以人为中心所产生的各种用户行为、用户业务活动和交易记录、用户社交数据，这些核心数据的相关性再加上可感知设备的智能数据采集就构成了一个完整的大数据生态环境。

1.3.3　大数据应用平台方案案例

大数据的应用在当前的互联网领域尤其以企业为主，企业成为大数据应用的主体。大数据真能改变企业的运作方式吗?答案毋庸置疑是肯定的。随着企业开始利用大数据，我们每天都会看到大数据新的奇妙的应用，帮助人们真正从中获益。大数据的应用已广泛深入我们生活的方方面面，涵盖医疗、交通、金融、教育、体育、零售等各行各业。

大数据应用的关键，也是其必要条件，就在于“IT”与“经营”的融合，当然，这里的经营的内涵可以非常广泛，小至一个零售门店的经营，大至一个城市的经营。以下是关于各行各业不同的组织机构在大数据应用案例的举例。如图 1-5 所示是大数据应用案例行业占比。

图 1-5　大数据应用案例行业占比

以下是《互联网周刊》发布的《大数据应用案例 TOP100》部分例举。

1. 深圳市儿童医院成功部署 IBM 集成平台与商业智能分析系统

IBM 利用其行业领先的大数据与分析技术，支持深圳市儿童医院搭建信息集成平台，整合原有分散在多系统中的海量数据，实现各部门的信息共享；同时通过商业智能分析对集成数据进行深入挖掘，为医院各部门人员的科学决策提供全面的辅助，提升医院的服务水平和管理能力。

2. Informatica 帮助紫金农商银行深挖数据价值

紫金农商银行 ODS 数据仓库项目建设使用 Informatica 产品完成数据的加载、清洗、转换工作显得尤为简单，图形化、流程化设计使维护人员能够快速、顺畅地操作，即使数据源结构发生变化，也不会像以前必须修改大量的程序代码，只需要在 PowerCenter 中配置一下即可。

3. 华为大数据一体机服务于北大重点实验室

经过大量的前期调查、比较和分析准备工作，北大重点实验室选择了华为基于高性能服务器 RH5885 V2 的 HANA 数据处理平台。HANA 提供的对大量实时业务数据进行快速查询和分析以及实时数据计算等功能，在很大程度上得益于华为 RH5885 V2 服务器的高可靠、高性能和高可用性的支撑。

4. IBM 携手汉端科技为飞鹤乳业打造全产业链可追溯体系

IBM、汉端科技与中国飞鹤乳业联合宣布，通过利用 IBM 业界领先的全面大数据与分析能力和汉端科技在商业智能领域丰富的行业经验，飞鹤乳业实现了产品的可追溯与食品安全的数字化管理，完成了系统数字化、透明化、服务化的升级。

5. 浪潮大数据平台大大提升了济南的警务工作能力

浪潮帮助济南公安局在搭建云数据中心的基础上构建了大数据平台，以开展行为轨迹分析、社会关系分析、生物特征识别、音视频识别、银行电信诈骗行为分析、舆情分析等多种大数据研判手段的应用，为指挥决策、各警种情报分析、研判提供支持，做到围绕治安焦点能够快速精确定位、及时全面掌握信息、科学指挥调度警力和社会安保力量迅速解决问题。

6. 英特尔携杭州诚道科技构建智能交通

面对大数据挑战，杭州市和杭州诚道科技有限公司紧密合作，部署了基于英特尔大数据解决方案的诚道重点车辆动态监管系统，通过集中的数据中心将全市卡口、电子警察、视频监控、流量检测设备、信号机、诱导设备等有效地连接起来，从交通案件侦破能力、交通警察对机动车辆的监管能力到利用关联车辆的数据分析能力，都得到了极大提升。

7. 步步高集团借 Oracle Exadata 大大提高了 IT 投资回报率

步步高集团采用 Oracle Exadata 数据库云服务器搭建信息化平台，凭借 Oracle Exadata 数据库云服务器的高扩展性、安全性和冗余性，步步高集团得以在该基础架构上运行一系列 Oracle 零售行业以及 Oracle 的应用软件。此外，基于 Oracle Exadata 的步步高 IT 新架构比传统架构拥有更好的性价比，最大限度地增加了 IT 的投资回报率。

8. 华为 Anti-DDoS 助阿里巴巴检测 DDoS 变革

阿里巴巴现网多个数据中心出口都部署了华为的 Anti-DDoS 解决方案，平均每天防护的 DDoS 攻击次数超过 100 次，每年达数万次，峰值防护的 DDoS 攻击流量超过 100Gbps。如今，DDoS 攻击在阿里巴巴安全工程师眼里已经习以为常，由华为 Anti-DDoS 方案自动调度进行清洗防护即可。“双十一”期间，华为 Anti-DDoS 方案一如既往地成功防护了多轮 DDoS 攻击事件，有力保障了阿里巴巴网络交易的顺畅平稳。

1.4 并行计算简介

并行计算或称平行计算是相对于串行计算来说的。它是一种一次可执行多个指令的算法，目的是提高计算速度以及通过扩大问题求解规模，解决大型而复杂的计算问题。所谓并行计算可分为时间上的并行和空间上的并行。时间上的并行就是指流水线技术，而空间上的并行则是指用多个处理器并发地执行计算。

并行计算的定义：并行计算（Parallel Computing）是指同时使用多种计算资源解决计算问题的过程，是提高计算机系统计算速度和处理能力的一种有效手段。它的基本思想是用多个处理器来协同求解同一问题，即将被求解的问题分解成若干个部分，各部分均由一个独立的处理机来并行计算。并行计算系统既可以是专门设计的、含有多个处理器的超级计算机，也可以是以某种方式互连的若干台的独立计算机构成的集群。通过并行计算集群完成数据的处理，再将处理的结果返回给用户。

为利用并行计算，通常计算问题表现为以下计算问题特征：

- 将工作分离成离散部分，有助于同时解决。
- 随时并及时地执行多个程序指令。
- 多计算资源下解决问题的耗时要少于单个计算资源下的耗时。

并行计算科学中主要研究的是空间上的并行问题。从程序和算法设计人员的角度来看，

并行计算又可分为数据并行和任务并行。一般来说，因为数据并行主要是将一个大任务化解成相同的各个子任务，比任务并行要容易处理。如图 1-6 所示为问题的并行求解过程。

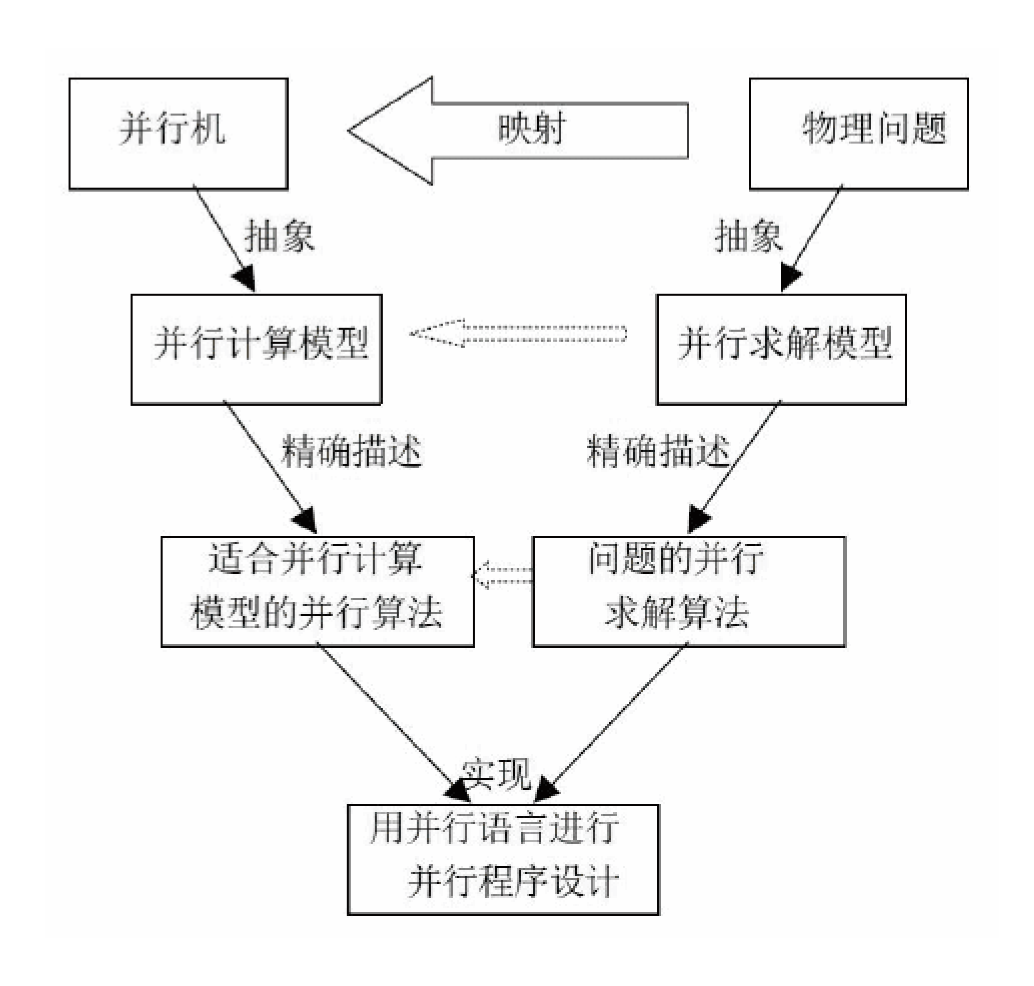


图 1-6　问题的并行求解过程

云计算的萌芽应该从计算机的并行化开始，并行机的出现是人们不满足于 CPU 摩尔定律的增长速度，希望把多个计算机并联起来，从而获得更快的计算速度。这是一种很简单也很朴素的实现高速计算的方法，这种方法后来被证明是相当成功的。

1.5 Hadoop 介绍

Hadoop 是一个由 Apache 基金会所开发的分布式系统基础架构。用户可以在不了解分布式底层细节的情况下，开发分布式程序，充分利用集群的威力进行高速运算和存储。

Hadoop 实现了一个分布式文件系统（Hadoop Distributed File System），简称 HDFS。HDFS 有高容错性的特点，并且设计用来部署在低廉的（low-cost）硬件上，而且它提供高吞吐量（high throughput）来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。HDFS 放宽（relax）了 POSIX 的要求，可以以流的形式访问（streaming access）文件系统中的数据。

Hadoop 的框架最核心的设计就是：HDFS 和 MapReduce。HDFS 为海量的数据提供了存储， MapReduce 则为海量的数据提供了计算。

Hadoop 是最受欢迎的在 Internet 上对搜索关键字进行内容分类的工具，但它也可以解决许多要求极大伸缩性的问题。例如，如果你要 grep 一个 10TB 的巨型文件，会出现什么情况？在传统的系统上，这将需要很长的时间；但是 Hadoop 在设计时就考虑到这些问题，采用并行执行机制，因此能大大提高效率。

Hadoop 原本来自于 Google 一款名为 MapReduce 的编程模型包。Google 的 MapReduce 框架可以把一个应用程序分解为许多并行计算指令，跨大量的计算节点运行非常巨大的数据集。使用该框架的一个典型例子就是在网络数据上运行的搜索算法。Hadoop 最初只与网页索

引有关，迅速发展成为分析大数据的领先平台。

目前有很多公司开始提供基于 Hadoop 的商业软件、支持、服务以及培训。Cloudera 是一家美国的企业软件公司，该公司在 2008 年开始提供基于 Hadoop 的软件和服务。GoGrid 是一家云计算基础设施公司，在 2012 年，该公司与 Cloudera 合作加速了企业采纳基于 Hadoop 应用的步伐。Dataguise 公司是一家数据安全公司，同样在 2012 年该公司推出了一款针对 Hadoop 的数据保护和风险评估。

Hadoop 是一个能够对大量数据进行分布式处理的软件框架，Hadoop 以一种可靠、高效、可伸缩的方式进行数据处理。

Hadoop 是可靠的，因为它假设计算元素和存储会失败，因此它维护多个工作数据副本，确保能够针对失败的节点重新分布处理。

Hadoop 是高效的，因为它以并行的方式工作，通过并行处理加快处理速度。

Hadoop 还是可伸缩的，能够处理 PB 级数据。

此外，Hadoop 依赖于社区服务，因此它的成本比较低，任何人都可以使用。

Hadoop 是一个能够让用户轻松架构和使用的分布式计算平台。用户可以轻松地在 Hadoop 上开发和运行处理海量数据的应用程序。它主要有以下几个优点：

- 高可靠性。Hadoop 按位存储和处理数据的能力值得人们信赖。
- 高扩展性。Hadoop 是在可用的计算机集簇间分配数据并完成计算任务的，这些集簇可以方便地扩展到数以千计的节点中。
- 高效性。Hadoop 能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此处理速度非常快。
- 高容错性。Hadoop 能够自动保存数据的多个副本，并且能够自动将失败的任务重新分配。
- 低成本。与一体机、商用数据仓库以及 QlikView、Yonghong Z-Suite 等数据集市相比，hadoop 是开源的，项目的软件成本因此会大大降低。

Hadoop 带有用 Java 语言编写的框架，因此运行在 Linux 生产平台上是非常理想的。Hadoop 上的应用程序也可以使用其他语言编写，比如 C++。

Hadoop 在大数据处理应用中广泛应用得益于其自身在数据提取、变形和加载（ETL）方面上的天然优势。Hadoop 的分布式架构，将大数据处理引擎尽可能地靠近存储，对例如像 ETL 这样的批处理操作相对合适，因为类似这样操作的批处理结果可以直接走向存储。Hadoop 的 MapReduce 功能实现了将单个任务打碎，并将碎片任务（Map）发送到多个节点上，之后再以单个数据集的形式加载（Reduce）到数据仓库里。图 1-7 给出了 Hadoop 分布式核心系统框架。

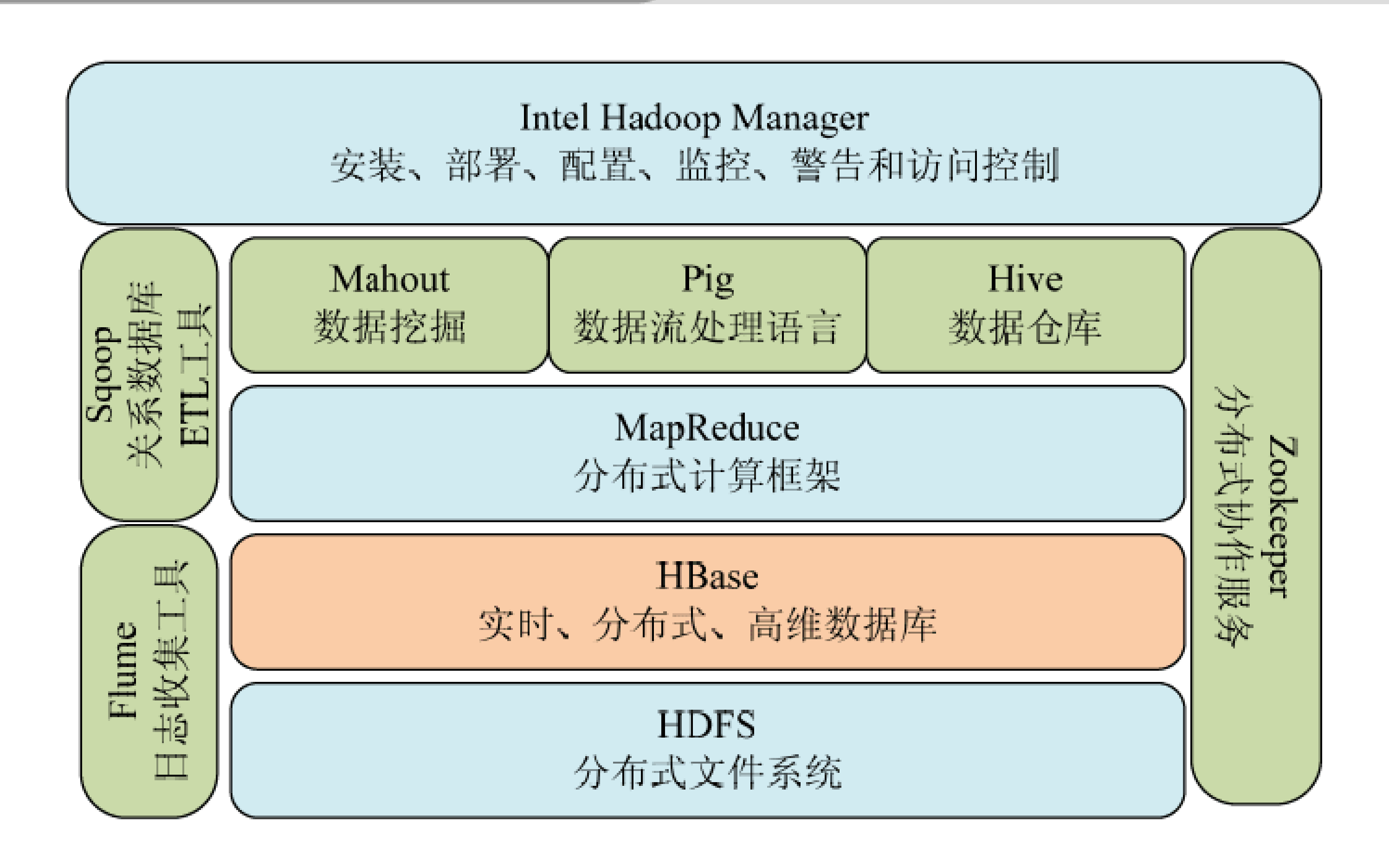


图 1-7　Hadoop 分布式核心系统框架

Hadoop 由许多元素构成。其最底部是 Hadoop Distributed File System（HDFS），它存储 Hadoop 集群中所有存储节点上的文件。HDFS（对于本文）的上一层是 MapReduce 引擎，该引擎由 JobTrackers 和 TaskTrackers 组成，以及数据仓库工具 Hive 和分布式数据库 HBase，基本涵盖了 Hadoop 分布式平台的所有技术核心。

1.6 本章小结

本章从大数据处理特点出发，概述了大数据来源、应用价值、技术特点和相关领域技术。在应用方面以案例为切合点介绍了不同场景下的大数据应用。最后简要描述了大数据处理的利器 Hadoop。

第 2 章

云计算时代

云计算一般被定义为在网络环境下计算资源的交付和使用方式，用户通过网络按需、易扩展的方式获得所需服务。要实现这个目标，需要 5 个最为关键的特征或者说条件来支撑：足够的宽带网络、资源“池化”、按需伸缩的弹性机制、服务自治（用户可以按需开通服务，后台自适应这种变化）、按使用量计算成本。

在标准模型中，云计算通常体现为 3 种服务交付模式：IaaS（基础设施即服务）、PaaS（平台即服务）、SaaS（软件即服务）。虽然也有其他模式，如 FaaS（框架即服务）、BaaS（流程即服务）等，但这些服务模式都可以纳入上述 3 种模式之中。这 3 种服务交付模式提供了计算资源从底层到顶层的交付。

2.1 云计算概述

云计算表面上看是服务的交付，其本质上又体现为能力的交付。例如：IaaS 本质上是云服务客户能配置和使用计算、存储和网络资源的一类云能力类型；PaaS 本质上是云服务客户能使用云服务提供者支持的编程语言和执行环境，部署、管理和运行客户创建或获取的应用的一类云能力类型；SaaS 本质上是云服务客户能使用云服务提供者的应用的一类云能力类型。云计算实现自来计算的根本原因就在于这些能力的形成。

2.1.1 云计算概念

云计算（cloudcomputing）是基于互联网的相关服务的增加、使用和交付模式，通常涉及通过互联网来提供动态易扩展且经常是虚拟化的资源。

人们过去利用计算资源主要依赖于独立的单台计算机，受制于物理机器资源的数量。正是基于这个原因，人们开始期望计算资源能够像自来水和电一样按需供应。而云计算的出现使自来计算变成现实，自来计算是人类的现实需要，是需求创造的结果。当然，从人类利用计算技术的趋势来看，云计算只是实现自来计算的一种模式，随着技术的进步，人们也许会发现新的实现自来计算的模式[6]。

云计算是一种商业计算模型，它将计算任务分布在大量计算机构成的资源池上，使用户能够按需获取计算力、存储空间和信息服务。

从技术上看，大数据与云计算的关系就像混合面的馒头一样密不可分。大数据必然无法用单台的计算机进行处理，必须采用分布式计算架构。它的特色在于对海量数据的挖掘，但

它必须依托云计算的分布式处理、分布式数据库、云存储和虚拟化技术。

按照部署主体的不同，云计算通常有 4 种部署方式：公有云、私有云、混合云和社区云。公有云就是由第三方云计算服务商部署的云计算平台，用户通过租用的方式使用它；私有云就是一个企业或机构建设为内部使用的云计算平台，这两种云是最常见的，也是基础的云部署方式。混合云和社区云更多是衍生的概念。当一个企业的私有云不能满足需要，或者出现业务起伏的情况，但又不值得去扩张云计算中心，那么就会租赁公有云部分资源使用，技术上已能够实现私有云和公有云的连接，这就是混合云。社区云是指云基础设施由若干个组织分享，以支持某个特定的社区。社区是指有共同诉求和追求的团体（例如使命、安全要求、政策或合规性考虑等）。和私有云类似，社区云可以是该组织或第三方负责管理，可以是场内服务，也可以是场外服务。上述是严格的概念，实际使用中又会出现很多复杂情况。例如，在一个公有云当中特设一个区域专为某一个企业服务，对企业而言是私有云，但同时它又是由公有云分割出来，这种情况就构成了虚拟数据中心。还有一种情况是，企业建设私有云无须自己建设机房，可以把自己的云平台完全托管在一个公共数据中心。这与前面虚拟数据中心又不同，属于私有云的托管。所以，用户可以根据实际情况来部署，而不需拘泥于形式[7,8]。

2.1.2 云计算发展简史

追根溯源，云计算与并行计算、分布式计算和网格计算不无关系，更是虚拟化、效用计算、SaaS、SOA 等技术混合演进的结果。本节总结回顾几十年来云计算一步步演变和发展历程中的点滴事件：

1959 年 6 月，ChristopherStrachey 发表虚拟化论文，虚拟化是今天云计算基础架构的基石。

1961 年，JohnMcCarthy 提出计算力和通过公用事业销售计算机应用的思想。

1962 年，J.C.R.Licklider 提出“星际计算机网络”设想。

1965 年，美国电话公司 WesternUnion 的一位高管提出建立信息公用事业的设想。

1983 年，太阳电脑（Sun Microsystems）提出“网络是电脑”（The Network is the Computer），2006 年 3 月，亚马逊（Amazon）推出弹性计算云（Elastic Compute Cloud，EC2）服务。

1984 年，Sun 公司的联合创始人 JohnGage 说出了“网络就是计算机”的名言，用于描述分布式计算技术带来的新世界，今天的云计算正在将这一理念变成现实。

1996 年，网格计算 Globus 开源网格平台起步。

1997 年，南加州大学教授 RamnathK.Chellappa 提出云计算的第一个学术定义“认为计算的边界可以不是技术局限，而是经济合理性。”

1998 年，VMware（威睿公司）成立并首次引入 X86 的虚拟技术。

1999 年，MarcAndreessen 创建 LoudCloud，是第一个商业化的 IaaS 平台。

1999 年，salesforce.com 公司成立，宣布“软件终结”革命开始。

2000 年，SaaS 兴起。

2004 年，Web2.0 会议举行，Web2.0 成为技术流行词，互联网发展进入新阶段。

2004 年，Google 发布 MapReduce 论文。Hadoop 就是 Google 集群系统的一个开源项目总称，主要由 HDFS、MapReduce 和 HBase 组成，其中 HDFS 是 GoogleFileSystem（GFS）的开源实现；MapReduce 是 GoogleMapReduce 的开源实现；HBase 是 GoogleBigTable 的开源实现。

2004 年，DougCutting 和 MikeCafarella 实现了 Hadoop 分布式文件系统（HDFS）和 Map-Reduce，Hadoop 成为非常优秀的分布式系统基础架构。

2005 年，Amazon 宣布 AmazonWebServices 云计算平台。

2006 年，Amazon 相继推出在线存储服务 S3 和弹性计算云 EC2 等云服务。

2006 年，Sun 推出基于云计算理论的“BlackBox”计划。

2006 年 8 月 9 日，Google 首席执行官埃里克·施密特（Eric Schmidt）在搜索引擎大会（SES San Jose 2006）首次提出“云计算”（Cloud Computing）的概念。Google“云端计算”源于 Google 工程师克里斯托弗·比希利亚所做的“Google 101”项目。

2007 年，Google 与 IBM 在大学开设云计算课程。

2007 年 3 月，戴尔成立数据中心解决方案部门，先后为全球 5 大云计算平台中的三个（包括 WindowsAzure、Facebook 和 Ask.com）提供云基础架构。

2007 年 7 月，亚马逊公司推出了简单队列服务（SimpleQueueService，SQS），这项服务使托管主机可以存储计算机之间发送的消息。

2007 年 10 月，Google 与 IBM 开始在美国大学校园，包括卡内基梅隆大学、麻省理工学院、斯坦福大学、加州大学柏克莱分校及马里兰大学等，推广云计算的计划，这项计划希望能降低分布式计算技术在学术研究方面的成本，并为这些大学提供相关的软硬件设备及技术支持（包括数百台个人电脑及 BladeCenter 与 System x 服务器，这些计算平台将提供 1600 个处理器，支持包括 Linux、Xen、Hadoop 等开放源代码平台）。而学生则可以通过网络开发各项以大规模计算为基础的研究计划。

2008 年 1 月 30 日，Google 宣布在台湾地区启动“云计算学术计划”，将与台湾台大、交大等学校合作，将这种先进的大规模、快速度的云计算技术推广到校园。

2008 年 2 月 1 日，IBM（NYSE: IBM）宣布将在中国无锡太湖新城科教产业园为中国的软件公司建立全球第一个云计算中心（Cloud Computing Center）。

2008 年 7 月 29 日，雅虎、惠普和英特尔宣布一项涵盖美国、德国和新加坡的联合研究计划，推出云计算研究测试床，推进云计算。该计划要与合作伙伴创建 6 个数据中心作为研究试验平台，每个数据中心配置 1400 个至 4000 个处理器。这些合作伙伴包括新加坡资讯通信发展管理局、德国卡尔斯鲁厄大学 Steinbuch 计算中心、美国伊利诺伊大学香槟分校、英特尔研究院、惠普实验室和雅虎。

2008 年 8 月 3 日，美国专利商标局网站信息显示，戴尔正在申请“云计算”（Cloud Computing）商标，此举旨在加强对这一未来可能重塑技术架构的术语的控制权。

2009 年 1 月，阿里软件在江苏南京建立首个“电子商务云计算中心”。

2009 年 4 月，VMware 推出业界首款云操作系统 VMwarevSphere4。

2009 年 7 月，Google 宣布将推出 ChromeOS 操作系统。

2009 年 7 月，中国首个企业云计算平台诞生（中化企业云计算平台）。

2009 年 9 月，VMware 启动 vCloud 计划构建全新云服务。

2009 年 11 月，中国移动云计算平台“大云”计划启动。

2010 年 1 月，HP 和微软联合提供完整的云计算解决方案。

2010 年 1 月，IBM 与松下达成迄今为止全球最大的云计算交易。

2010 年 1 月，Microsoft 正式发布 MicrosoftAzure 云平台服务。

2010 年 3 月 5 日，Novell 与云安全联盟（CSA）共同宣布一项供应商中立计划，名为“可信任云计算计划（Trusted Cloud Initiative）”。

2010 年 4 月，英特尔在 IDF 上提出互联计算，图谋用 X86 架构统一嵌入式、物联网和云计算领域。

2010 年，微软宣布其 90%员工将从事云计算及相关工作。

2010 年 4 月，戴尔推出源于 DCS 部门设计的 PowerEdgeC 系列云计算服务器及相关服务。

2010 年 7 月，美国国家航空航天局和包括 Rackspace、AMD、Intel、戴尔等支持厂商共同宣布“OpenStack”开放源代码计划，微软在 2010 年 10 月表示支持 OpenStack 与 Windows Server 2008 R2 的集成；而 Ubuntu 已把 OpenStack 加至 11.04 版本中。

2011 年 2 月，思科系统正式加入 OpenStack，重点研制 OpenStack 的网络服务。

2.1.3 云计算实现机制

由于云计算分为 IaaS、PaaS 和 SaaS 三种类型，图 2-1 给出了云计算技术体系结构，这个体系结构概括了不同解决方案的主要特征，每一种方案或许只实现了其中部分功能，或许也还有部分相对次要功能尚未概括进来。

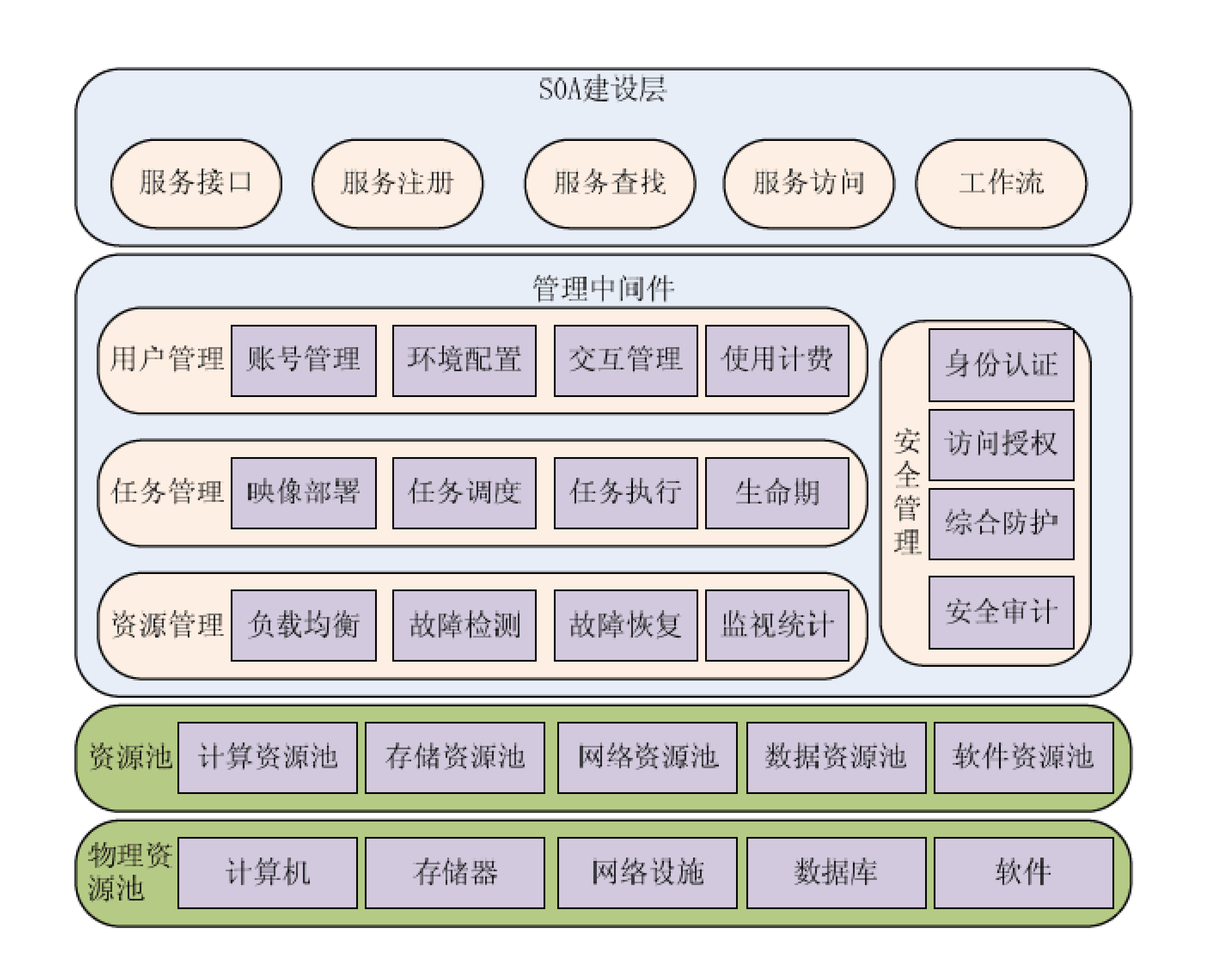


图 2-1　云计算技术体系结构

云计算技术体系结构分为 4 层：物理资源层、资源池层、管理中间件层和 SOA 构建层，如图 2-1 所示。物理资源层包括计算机、存储器、网络设施、数据库和软件等；资源池层是将大量相同类型的资源构成同构或接近同构的资源池，如计算资源池、数据资源池等。构建

资源池更多是物理资源的集成和管理工作；管理中间件负责对云计算的资源进行管理，并对众多应用任务进行调度，使资源能够高效、安全地为应用提供服务；SOA 构建层将云计算能力封装成标准的 Web Services 服务，并纳入到 SOA 体系进行管理和使用，包括服务注册、查找、访问和构建服务工作流等。管理中间件和资源池层是云计算技术的最关键部分，SOA 构建层的功能更多依靠外部设施提供。

云计算的管理中间件负责资源管理、任务管理、用户管理和安全管理等工作。资源管理负责均衡地使用云资源节点、检测节点的故障并试图恢复或屏蔽之，并对资源的使用情况进行监视统计；任务管理负责执行用户或应用提交的任务，包括完成用户任务映象(Image)的部署和管理、任务调度、任务执行、任务生命期管理等；用户管理是实现云计算商业模式的一个必不可少的环节，包括提供用户交互接口、管理和识别用户身份、创建用户程序的执行环境、对用户的使用进行计费等；安全管理保障云计算设施的整体安全，包括身份认证、访问授权、综合防护和安全审计等。

基于上述体系结构，本节以 IaaS 云计算为例，简述云计算的实现机制，如图 2-2 所示[9]。用户交互接口向应用以 Web Services 方式提供访问接口，获取用户需求。服务目录是用户可以访问的服务清单。系统管理模块负责管理和分配所有可用的资源，其核心是负载均衡。配置工具负责在分配的节点上准备任务运行环境。监视统计模块负责监视节点的运行状态，并完成用户使用节点情况的统计。用户交互接口允许用户从目录中选取并调用一个服务。该请求传递给系统管理模块后，它将为用户分配恰当的资源，然后调用配置工具来为用户准备运行环境。

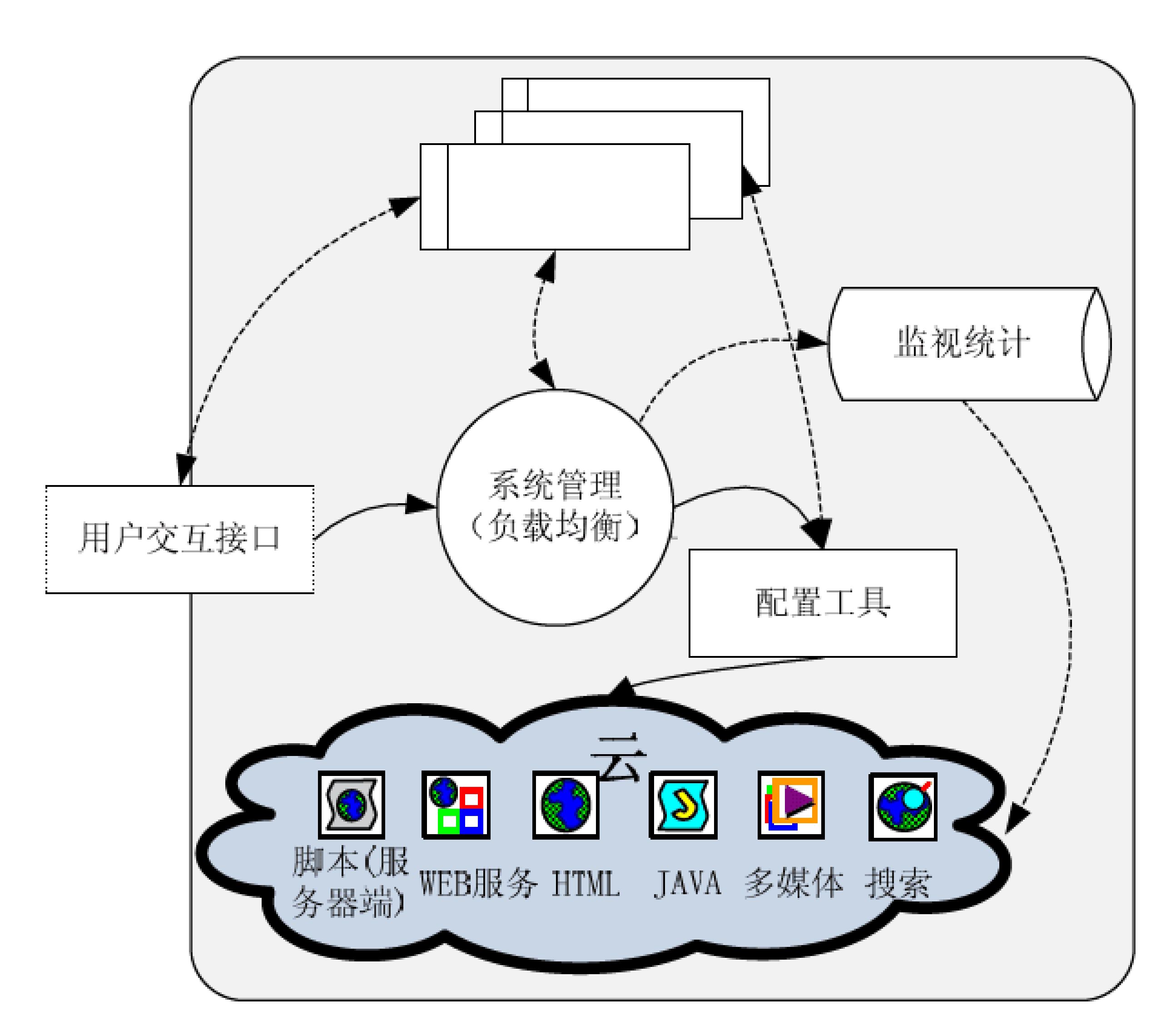


图 2-2　简化 IaaS 实现机制

2.1.4　云计算服务形式

云计算服务是指将大量用网络连接的计算资源统一管理和调度，构成一个计算资源池向用户按需服务。用户通过网络以按需、易扩展的方式获得所需资源和服务。它足够智能，能

够根据你的位置、时间、偏好等信息，实时地对你的需求做出预期。

云服务的商业模式是通过繁殖大量创业公司提供丰富的个性化产品，以满足市场上日益膨胀的个性化需求。其繁殖方式是为创业公司提供资金、推广、支付、物流、客服一整套服务，把自己的运营能力像水和电一样让外部随需使用。

云服务提供商，为中小企业搭建信息化所需要的所有网络基础设施及软件、硬件运作平台，并负责所有前期的实施、后期的维护等一系列服务，企业无须购买软硬件、建设机房、招聘 IT 人员，只需前期支付一次性的项目实施费和定期的软件租赁服务费，即可通过互联网享用信息系统。

云计算服务应该具备以下几条特征：

（1）随需自助服务。
（2）随时随地用任何网络设备访问。
（3）多人共享资源池。
（4）快速重新部署灵活度。
（5）可被监控与量测的服务。
（6）基于虚拟化技术快速部署资源或获得服务。
（7）减少用户终端的处理负担。
（8）降低了用户对于 IT 专业知识的依赖。

云架构数据中心具有多种优点，逐渐成为企业应用的大趋势，企业选择采用云架构数据中心，应考虑服务提供商是否符合以下三大要点。

首先，云架构数据中心服务提供商必须具有优秀的云端骨干网络，确保数据传输的顺畅及安全。如第一线集团，既可提供 MPLS 专线网络服务，又可同时提供云架构数据中心服务。对于许多跨地域的企业来说，除需要数据服务之外，更需要解决与分布世界各地的分支机构通信的问题，故稳定的网络质量一定是需要重点考量的。

其次，云架构数据中心是否有足够的弹性，是否可以让用户随时选择调配不同的云资源，以符合自身的使用需要。以金融业为例，每日股票市场的交易时段最为繁忙，云端数据中心有时候需要在短时间内提升工作性能，以应付各种突如其来的需要。

最后，服务提供商有没有提供简单、易用的软件，让企业人员安在办公室之中，也可自行掌握数据中心的所有运作，如同管理实体数据中心一样。

2.1.5 云计算时代的数据库 NoSQL

随着云计算时代的到来，各种类型的互联网应用层出不穷，对与此相关的数据模型、分布式架构、数据存储等数据库相关的技术指标也提出了新的要求。虽然传统的关系型数据库已在数据存储方面占据了不可动摇的地位，但由于其天生的限制，已经越来越无法满足云计算时代对数据扩展、读写速度、支撑容量以及建设和运营成本的要求。云计算时代对数据库技术提出了新的需求，主要表现在以下几个方面：

（1）海量数据处理：对类似搜索引擎和电信运营商级的经营分析系统这样大型的应用而言，需要能够处理 PB 级的数据，同时应对百万级的流量。

（2）大规模集群管理：分布式应用可以更加简单地部署、应用和管理。

（3）低延迟读写速度：快速的响应速度能够极大地提高用户的满意度。

（4）建设及运营成本：云计算应用的基本要求是希望在硬件成本、软件成本以及人力成本方面都有大幅度的降低。

关系型数据库遇到上述难以克服的瓶颈，与此同时，它的很多主要特性在**云计算**应用中却往往无用武之地，例如：数据库事务一致性、数据库的写实时性和读实时性、复杂的 SQL 查询特别是多表关联查询。因此，传统的关系型数据库已经无法独立应付云计算时代的各种应用。

关系型数据库越来越无法满足云计算的应用场景，为了解决此类问题，非关系型数据库应运而生，由于在设计上和传统的关系型数据库相比有了很大的不同，所以此类数据库被称为“NoSQL（Not only SQL）”系列数据库。与关系型数据库相比，它们非常关注对数据高并发读写和海量数据的存储，在架构和数据模型方面做了简化，而在扩展和并发等方面做了增强。目前，主流的 NoSQL 数据库包括 BigTable、HBase、Cassandra、SimpleDB、CouchDB、MongoDB 以及 Redis 等。NoSQL 常用数据模型包括以下 3 种：

1. Column-oriented（列式）

列式主要使用 Table 这样的模型，但是它并不支持类似 Join 这样多表的操作，它的主要特点是在存储数据时，主要围绕着“列（Column）”，而不是像传统的关系型数据库那样根据“行（Row）”进行存储，也就是说，属于同一列的数据会尽可能地存储在硬盘同一个页中，而不是将属于同一个行的数据存放在一起。这样做的好处是，对于很多类似数据仓库的应用，虽然每次查询都会处理很多数据，但是每次所涉及的列并没有很多。使用列式数据库，将会节省大量 I/O，并且大多数列式数据库都支持 Column Family 这个特性，能将多个列并为一个小组。这样做的好处是能将相似列放在一起存储，提高这些列的存储和查询效率。总体而言，这种数据模型的优点是比较适合汇总和数据仓库这类应用。

2. Key-value

虽然 Key-value 这种模型和传统的关系型相比较简单，有点类似常见的 HashTable，一个 Key 对应一个 Value；但是它能提供非常快的查询速度、大的数据存放量和高并发操作，非常适合通过主键对数据进行查询和修改等操作，虽然不支持复杂的操作，但是可以通过上层的开发来弥补这个缺陷。

3. Document（文档）

在结构上，Document 和 Key-value 是非常相似的，也是一个 Key 对应一个 Value；但是这个 Value 主要以 JSON 或者 XML 等格式的文档来进行存储，是有语义的，并且 Document DB 一般可以对 Value 创建 Secondary Index 来方便上层的应用，而这点是普通 Key-Value DB 所无法支持的。

云计算主要常见的有两类场景：需要低延迟和高并发的读写能力，数据量虽大，但不超过 TB 级别，大部分现在使用 RDBMS 的 Web 应用基本上都属于这一类，类似传统的 OLTP（联机事务处理）；海量数据的存储和操作，如 PB 级别的，这方面的例子有传统的数据仓库、Google 海量的 Web 页面和图片存储等，类似传统的 OLAP（联机分析处理）。目前，业

界还没有一款数据库能同时适应上述多种云计算场景的 NoSQL 数据库。

2.2 云计算发展动力源泉

云计算作为新一代信息技术的重要代表，正在深刻地改变着信息产业的发展格局，推动生产、生活方式发生着重大的革命性变化。

微软早在 16 年前就开始了云计算的研发工作，用户已经很熟悉的 hotmail、XBox360 live、Office365 等产品就是典型的云计算的应用。微软目前 90%的研发团队在从事与云计算相关的工作。

在云计算领域，各国几乎都处在同一个起跑线，在欧美等西方国家，云计算的发展更多的是市场导向。

2015 年的《政府工作报告》中提出，要制定“互联网+”行动计划，推动移动互联网、云计算、大数据、物联网等与现代制造业结合。这既是对云计算推动产业转型升级重要作用的充分肯定，也为今后云计算产业的发展指明了方向。

当前，云计算发展面临三大发展机遇：

（1）云计算作为 IT 产业发展的重要方向。

（2）应用驱动是云计算发展的重要动力源泉。正是这样规模最大、应用最复杂、需求最有特点的市场，培育出了若干的云计算应用。而复杂的系统和庞大的应用需求为发展云计算带来了很好的先机。

（3）软件技术架构正在发生重大变革。以云计算、物联网、移动互联网、大数据为代表的新技术发展和应用，使得传统软件架构已经被颠覆，新的面向服务的架构和应用正在成为潮流和趋势。

2.3 云计算技术分析

云计算是大规模分布式计算技术及其配套商业模式演进的产物，它的发展主要有赖于虚拟化、分布式数据存储、数据管理、编程模式、信息安全等各项技术、产品的共同发展。近些年来，托管、后向收费、按需交付等商业模式的演进也加速了云计算市场的转折。云计算不仅改变了信息提供的方式，也颠覆了传统 ICT 系统的交付模式。与其说云计算是技术的创新，不如说云计算是思维和商业模式的转变。

2.3.1 编程模式

作为一种新兴的计算模式，云计算以互联网服务和应用为中心，其背后是大规模集群和海量数据，新的场景需要新的编程模型来支撑。云计算场景下，新的编程模型要能够方便快速地分析和处理海量数据，并提供安全、容错、负载均衡、高并发和可伸缩性等机制。

为了能够低成本高效率地处理海量数据，主要的互联网公司都在大规模集群系统上研发

了分布式编程系统，使普通开发人员可以将精力集中于业务逻辑上，不用关注分布式编程的底层细节和复杂性，从而降低普通开发人员编程处理海量数据并充分利用集群资源的难度。

目前云计算编程模式的研究在工业界和学术界方兴未艾，各种新思想和新技术不断出现。本节下面重点介绍几种有代表性的编程模型。

1. MapReduce

MapReduce 是 Google 公司的 Jeff Dean 等人提出的编程模型，用于大规模数据的处理和生成。从概念上讲，MapReduce 处理一组输入的 key/value 对（键值对），产生另一组输出的键值对。当前的软件实现是指定一个 Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的 Reduce（化简）函数，用来保证所有映射的键值对中的每一个共享相同的键组。程序员只需要根据业务逻辑设计 Map 和 Reduce 函数，具体的分布式、高并发机制由 MapReduce 编程系统实现。

MapReduce 在 Google 得到了广泛应用，包括反向索引构建、分布式排序、Web 访问日志分析、机器学习、基于统计的机器翻译、文档聚类等。

Hadoop——作为 MapReduce 的开源实现——得到了 Yahoo!、Facebook、IBM 等大量公司的支持和应用。

2. Dryad

Dryad 是 Microsoft 设计并实现的允许程序员使用集群或数据中心计算资源的数据并行处理编程系统。从概念上讲，一个应用程序表示成一个有向无环图(Directed Acyclic Graph，DAG)。顶点表示计算，应用开发人员针对顶点编写串行程序，顶点之间的边表示数据通道，用来传输数据，可采用文件、TCP 管道和共享内存的 FIFO 等数据传输机制。Dryad 类似 UNIX 中的管道。如果把 UNIX 中的管道看成一维，即数据流动是单向的，每一步计算都是单输入单输出，整个数据流是一个线性结构；那么 Dryad 可以看成是二维的分布式管道，一个计算顶点可以有多个输入数据流，处理完数据后，可以产生多个输出数据流，一个 Dryad 作业是一个 DAG。Dryad 作业结构图如图 2-3 所示。

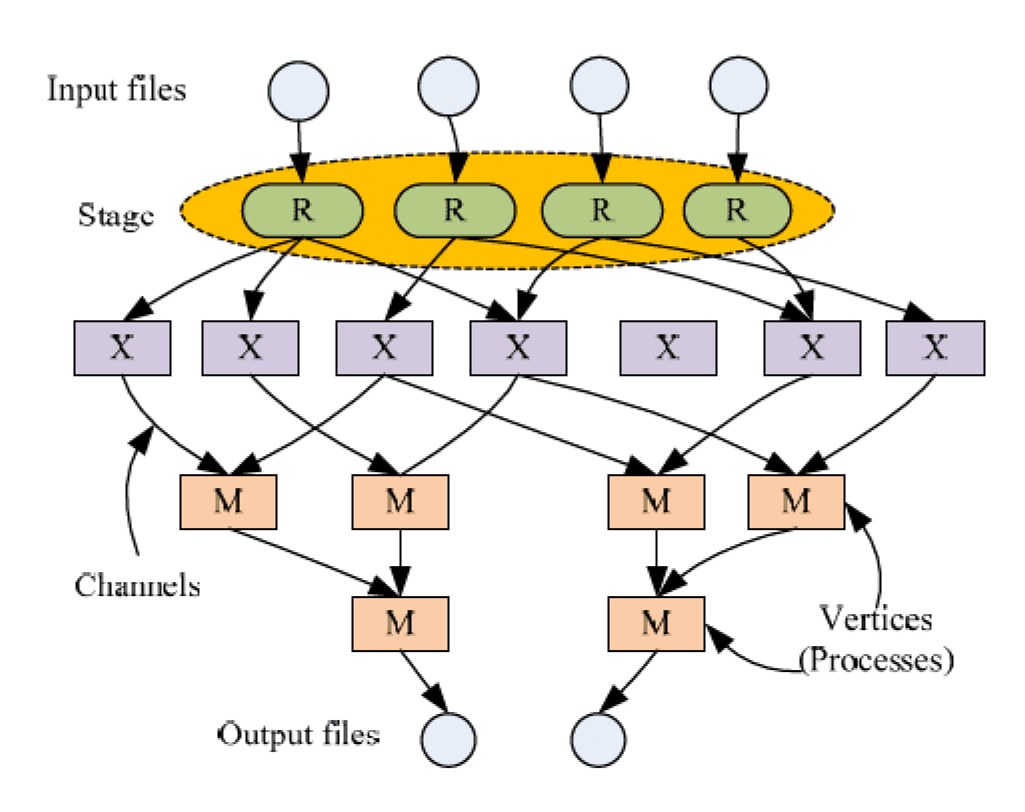


图 2-3　Dryad 作业结构图

Dryad 是针对运行 Windows HPC Server 的计算机集群设计的，是 Microsoft 构建云计算基础设施的核心技术之一。

3. Pregel

Pregel 是 Google 提出的一个面向大规模图计算的通用编程模型。许多实际应用中都涉及大型的图算法，典型的如网页链接关系、社交关系、地理位置图、科研论文中的引用关系等，有的图规模可达数十亿的顶点和上万亿的边。Pregel 编程模型就是为了对这种大规模图进行高效计算而设计。

4. All-Pairs

All-Pairs 是从科学计算类应用中抽象出来的一种编程模型。从概念上讲，All-Pairs 解决的问题可以归结为求集合 A 和集合 B 的笛卡尔积。All-Pairs 模型典型应用场景是比较两个图片数据集中任意两张图片的相似度。典型的 All-Pairs 计算包括四个阶段：首先对系统建模求最优的计算节点个数；随后向所有的计算节点分发数据集；接着调度任务到响应的计算节点上运行；最后收集计算结果。

5. Sawzall

Sawzall 是 Google 建立在其 MapReduce 编程模型之上的查询语言，Sawzall 的典型任务是在成百或上千台机器上并发操作上百万条记录。整个计算分为两个阶段：过滤阶段（相当于 Map 阶段）和聚合阶段（相当于 Reduce 阶段），并且过滤和聚合均可以在大量的分布式节点上并行执行。Sawzall 程序实现非常简洁，据统计一个完成相同功能的 MapReduce C++ 程序代码量是 Sawzall 程序代码量的 10~20 倍。

6. FlumeJava

FlumeJava 是一个建立在 MapReduce 之上的 Java 库，适合由多个 MapReduce 作业拼接在一起的复杂计算场景使用。FlumeJava 能简单地开发、测试和执行数据并行管道。

FlumeJava 库位于 MapReduce 等原语的上层，在允许用户表达计算和管道（Pipeline）信息的前提下，通过自动的优化机制后调用 MapReduce 等底层原语进行执行。Flumejava 首先优化执行计划，然后基于底层的原语来执行优化了的操作。

7. DryadLINQ

DryadLINQ 是 Microsoft 的高级编程语言。DryadLINQ 结合了 Microsoft 的两个重要技术：Dryad 语言和查询语言 LINQ。DryadLINQ 使用和 LINQ 相同的编程模型，并扩展了少量操作符和数据类型以适用于分布式计算。DryadLINQ 程序是一个顺序的 LINQ 代码，对数据集做任何无副作用的操作，编译器会自动地将数据并行的部分翻译成执行计划，交给底层的 Dryad 完成计算。

8. Pig Latin

Pig Latin 是 Yahoo!研发的运行在其 Pig 系统上的数据流语言。Pig 是高层次的声明式 SQL 与低层次过程式的 MapReduce 之间的折中。Pig 系统将 Pig Latin 程序编译成一组 Hadoop（MapReduce 的开源实现）作业，然后进行执行。Pig 不仅提供了常见的数据处理操作，包括加载、存储、过滤、分组、排序和连接等，同时 Pig 还提供了丰富的数据模型，支持原子类型、字典、元组等数据结构，以及嵌套操作。

随着商业智能分析、社交网络分析、在线推荐、数据挖掘、机器学习等应用的普及和深

入，海量数据处理的应用领域越来越呈现出多样化的趋势。而这些应用领域的问题可以抽象为结构化数据处理、大规模图计算、迭代计算等多种不同类型的计算。这些不同的问题领域适合采用不同的编程模型，不存在能解决所有数据密集型应用的通用编程模型。可以预见，会不断出现新的编程模型来解决领域相关的应用。

2.3.2　海量数据云存储技术

随着 Internet 技术的快速发展，数据量呈现出爆炸式增长，对所需的存储系统有更高的要求——更大存储容量、高性能、高安全级别、高智能化等，传统的 SAN/NAS 存储技术已无法处理 PB/EB 级海量数据，存在容量、性能、扩展和费用等上的瓶颈。云存储是云计算概念上的延伸和发展，专注于解决云计算中海量数据的存储挑战，它不仅能给云计算服务提供专业的存储解决方案，而且还可以独立地发布存储服务。云存储是综合分布式文件系统、集群应用和网络等技术，通过软件让网络中存在的大量的、不同类型的存储设备协同工作，共同对外提供数据存储和业务访问功能的一个系统[10]。

云存储不是存储，而是服务。就如同云状的广域网和互联网一样，云存储对使用者来讲，不是指某一个具体的设备，而是指一个由许许多多个存储设备和服务器所构成的集合体。使用者使用云存储，并不是使用某一个存储设备，而是使用整个云存储系统带来的一种数据访问服务。所以严格来讲，云存储不是存储，而是一种服务。

云存储系统中的所有设备对使用者来讲都是完全透明的，任何地方的任何一个经过授权的使用者都可以通过一根接入线缆与云存储连接，对云存储进行数据访问。云存储系统的结构模型由下面 4 层组成。

1. 存储层

存储层是云存储最基础的部分。云存储中的存储设备往往数量庞大且多分布在不同地域，彼此之间通过广域网、互联网或者 FC 光纤通道网络连接在一起。存储设备之上是一个统一存储设备管理系统，可以实现存储设备的逻辑虚拟化管理、多链路冗余管理，以及硬件设备的状态监控和故障维护。

2. 基础管理层

基础管理层是云存储最核心的部分，也是云存储中最难以实现的部分。基础管理层通过集群、分布式文件系统和网格计算等技术，实现云存储中多个存储设备之间的协同工作，使多个存储设备可以对外提供同一种服务，并提供更大更强更好的数据访问性能。

CDN 内容分发系统、数据加密技术保证云存储中的数据不会被未授权的用户所访问，同时，通过各种数据备份和容灾技术和措施可以保证云存储中的数据不会丢失，保证云存储自身的安全和稳定。

3. 应用接口层

应用接口层是云存储最灵活多变的部分。不同的云存储运营单位可以根据实际业务类型开发不同的应用服务接口，提供不同的应用服务。比如视频监控应用平台、IPTV 和视频点播应用平台、网络硬盘引用平台、远程数据备份应用平台等。

4. 访问层

任何一个授权用户都可以通过标准的公用应用接口来登录云存储系统，享受云存储服务。云存储运营单位不同，云存储提供的访问类型和访问手段也不同。

云存储的核心是应用软件与存储设备相结合，通过应用软件来实现存储设备向存储服务的转变。

本节以阿里云存储系统为例介绍云存储解决方案。表格存储（Table Store）是构建在阿里云飞天分布式系统之上的 NoSQL 数据存储服务，提供海量结构化数据的存储和实时访问。表格存储以实例和表的形式组织数据，通过数据分片和负载均衡技术，实现规模上的无缝扩展。应用通过调用表格存储 API / SDK 或者操作管理控制台来使用表格存储服务。表格存储 Table Store，是一个即开即用，支持高并发、低延时、无限容量的 Nosql 数据存储服务。

图 2-4 为阿里云存储方案框架，表格存储 Table Store + 云服务器 ECS+ E-MapReduce+ 大数据计算服务 ODPS。

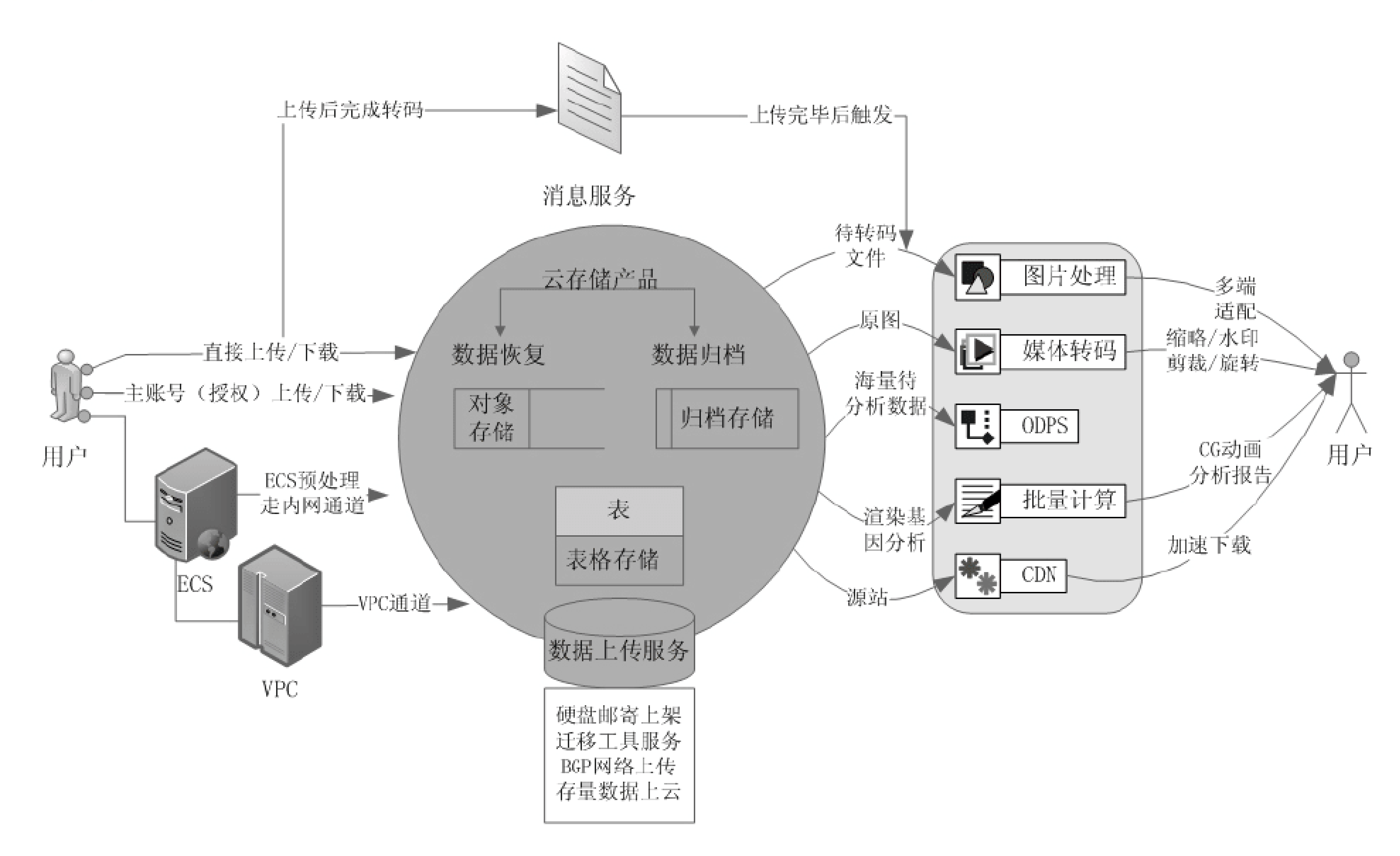


图 2-4 阿里云存储方案框架

2.3.3 海量数据管理技术

大数据（Big Data）对传统的数据管理技术带来了巨大挑战。目前围绕海量数据管理的热点问题研究方向是着重研究数据抽取与集成、数据存储与处理、信息检索、语义计算、数据隐私保护等方面技术[11,12]，主要包括：

（1）研究信息抽取及集成技术。侧重 Web 信息抽取和集成（包括 Deep Web、网页理解）。

（2）研究海量数据存储、分析及挖掘技术。重点研究大规模图数据（包括语义网数据）

等的存储、检索及挖掘技术。

（3）研究信息检索技术，包括语义检索、上下文感知的信息检索技术等。

（4）研究语义网与语义计算技术，包括 Web 数据规范表达、知识表达及大规模知识库推理、知识可视化技术等。

（5）数据隐私保护技术。研究面向隐私保护的多源海量数据挖掘与查询。

云系统为开发商和用户提供了简单通用接口，使得开发商将注意力集中在软件本身，无须考虑底层架构。云系统根据用户的请求动态分配计算资源。图 2-5 为云系统资源索引架构图。

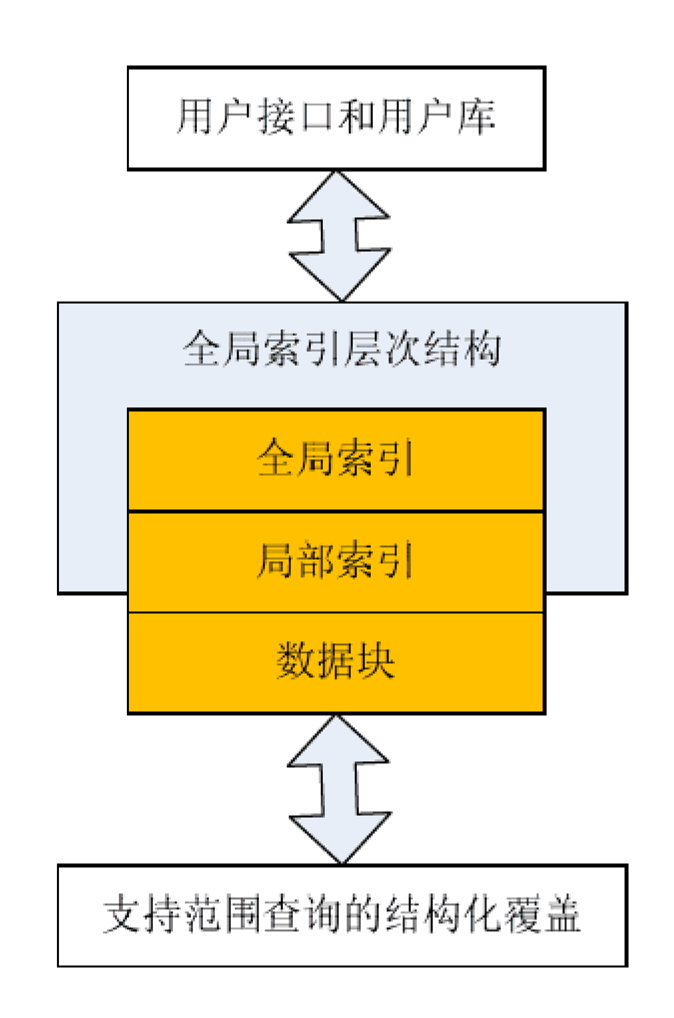


图 2-5　云系统资源索引架构图

该框架中，结构化覆盖网络形式组织处理节点，每个节点构建本体索引以加速数据访问。一个全局索引通过在覆盖网络中选择和发布一个本地索引分配来建立。目前这种方法在 Amazon EC2 上实践证明了有效性和可行性。

2.3.4　虚拟化技术

作为“智慧的信息技术”的重要组成部分，虚拟化与云计算已成为当今信息产业领域最受瞩目的新兴概念。但仍有许多人对虚拟化和云计算感觉很模糊，认为虚拟化就是云计算[13,14]。虚拟化技术是云计算的关键技术。云计算把计算当作公用资源，而不是一个具体的产品或者是技术。

云计算将各种 IT 资源以服务的方式通过互联网交付给用户，然而虚拟化本身并不能给用户提供自服务层。没有自服务层，就不能提供计算服务。云计算模型允许终端用户自行提供自己的服务器、应用程序和包括虚拟化等其他的资源，这反过来又能使企业最大程度的处理自身的计算资源，但这仍需要系统管理员为终端用户提供虚拟机。

虚拟化是指计算机元件在虚拟的基础上而不是真实的基础上运行。虚拟化技术可以扩大硬件的容量，简化软件的重新配置过程。CPU 的虚拟化技术可以单 CPU 模拟多 CPU 并行，允许一个平台同时运行多个操作系统，并且应用程序都可以在相互独立的空间内运行而互不

影响，从而显著提高计算机的工作效率。虚拟化实现了 IT 资源的逻辑抽象和统一表示，在大规模数据中心管理和解决方案交付方面发挥着巨大的作用，是支撑云计算伟大构想的最重要的技术基石。

虽然虚拟化和云计算并非是捆绑技术，但二者可以通过优势互补为用户提供更优质的服务。云计算方案使用虚拟化技术使整个 IT 基础设施的资源部署更灵活。反过来，虚拟化方案也可以引入云计算的理念，为用户提供按需使用的资源和服务。在一些特定业务中，云计算和虚拟化是分不开的，只有同时应用两项技术，服务才能顺利开展。表 2-1 给出了出几种虚拟化软件性能对比。

表 2-1　几种虚拟化软件性能对比

Attribute	Zones	Xen	KVM
CPU Performance	high	high (with CPU support)	high (with CPU support)
CPU Allocation	flexible (FSS + “bursting”)	fixed to VCPU limit	fixed to VCPU limit
I/O Throughput	high (no intrinsic overhead)	low or medium (with paravirt)	low or medium (with paravirt)
I/O Latency	low (no intrinsic overhead)	some (I/O proxy overhead)	some (I/O proxy overhead)
Memory Access Overhead	none	some (EPT/NPT or shadow page tables)	some (EPT/NPT or shadow page tables)
Memory Loss	none	some (extra kernels; page tables)	some (extra kernels; page tables)
Memory Allocation	flexible (unused guest memory used for file system cache)	fixed (and possible double-caching)	fixed (and possible double-caching)
Resource Controls	many (depends on OS)	some (depends on hypervisor)	most (OS + hypervisor)
Observability: from the host	highest (see everything)	low (resource usage, hypervisor statistics)	medium (resource usage, hypervisor statistics, OS inspection of hypervisor)
Observability: from the guest	medium (see everything permitted, incl. some physical resource stats)	low (guest only)	low (guest only)
Hypervisor Complexity	low (OS partitions)	high (complex hypervisor)	medium
Different OS Guests	usually no (sometimes possible with syscall translation)	yes	yes

表中的三列代表三种不同的类型：操作系统虚拟化（Zones）、硬件虚拟化的类型 1（Xen）和类型 2（KVM）。图 2-6 表示 KVM 寄居架构（Linux 内核）。

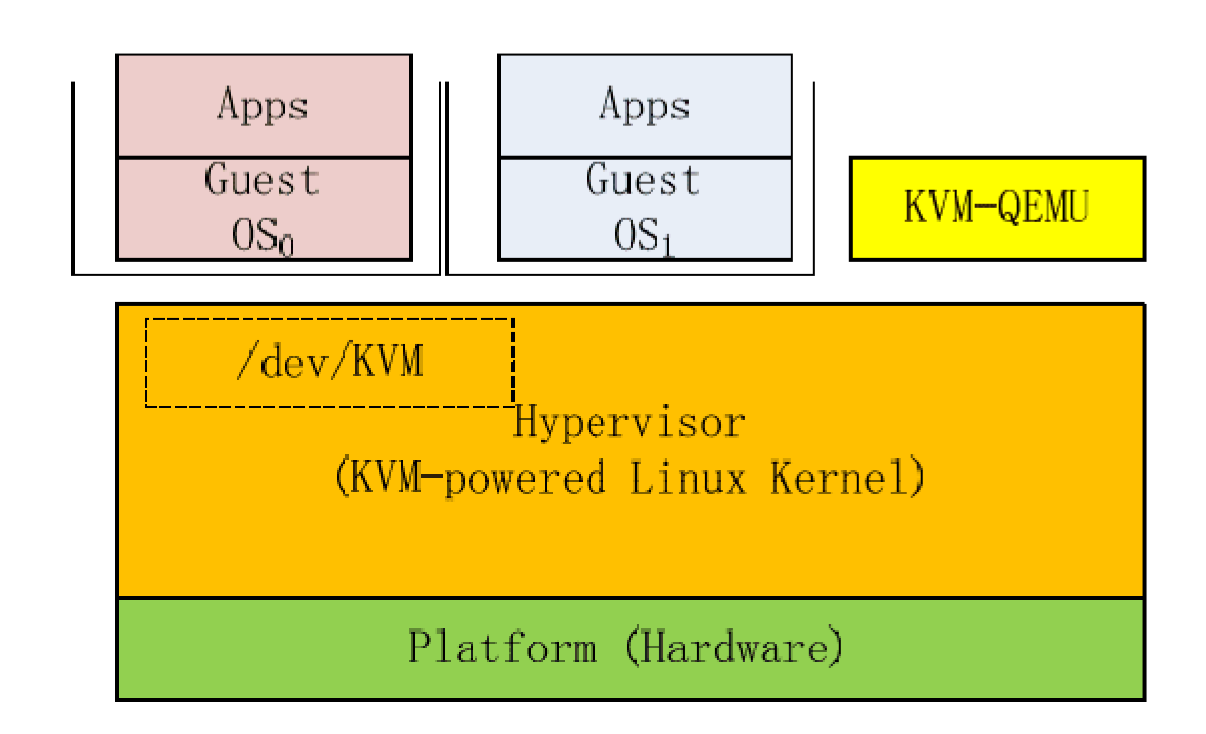


图 2-6　KVM 寄居架构（Linux 内核）

KVM 重用了整个 Linux I/O 协议栈，所以 KVM 的用户自然就获得了最新的驱动和 I/O 协议栈的改进。

2.3.5　分布式计算

分布式计算是利用互联网上计算机的中央处理器的闲置处理能力来解决大型计算问题的一种计算方法，和集中式计算是相对的。随着计算技术的发展，有些应用如果采用集中式计算，需要非常巨大的计算能力和耗费相当长的时间才能完成，分布式计算将该应用分解成许多小的部分，分配给多台计算机进行处理。这样可以节约整体计算时间，大大提高计算效率。分布式计算比起其他算法具有以下几个优点：

- 稀有资源可以共享。
- 通过分布式计算可以在多台计算机上平衡计算负载。
- 可以把程序放在最适合运行它的计算机上。

其中，共享稀有资源和平衡负载是计算机分布式计算的核心思想之一。实际上，网格计算就是分布式计算的一种。如果我们说某项工作是分布式的，那么，参与这项工作的一定不只是一台计算机，而是一个计算机网络，显然这种“蚁群”的方式将具有很强的数据处理能力。网格计算的实质就是组合与共享资源并确保系统安全。

2.3.6　云监测技术

云计算在受到各界广泛关注的同时，云计算提供的服务质量也越来越被重视。云平台的可靠性是提供给上层的一切服务质量的基础，如何提高云平台的可靠性是服务提供商重点关注的问题。将监控机制引入云平台是提高云平台可靠性和服务质量的有效途径。利用监控机制可以对异常状况实现预警，防患于未然。同时在故障发生之后可以通知平台管理人员，在第一时间进行处理。国外对监控技术的研究相比国内较早，已经有成熟的产品。为提高云计算平台的可靠性，保证服务质量，必须在云计算平台中引入监控机制来实时了解平台的运行状况，在平台出现异常时能起到预警的作用[15-22]。

如美国洛斯拉莫斯国家高级计算机实验室开发的 Supermon[23]集群监控系统，利用每个节点上的信息收集程序（mon）收集节点的状态信息，数据收集器（Supermon）汇总 mon 收集

的数据进行处理。虽然 Supermon 采用层次结构，但是由于系统中只有一个 Supermon 节点，当节点规模变大时，系统的性能呈线性下降，而且 Supermon 是系统的单一失效点，降低了系统的可靠性。印度高性能计算开发中心研发的 PARMON[24]利用 C/S 模式实现集群的监控，集群中的节点安装 parmon_server 程序，监控端利用 armon_client 获取集群节点数据，并通过 parmon_client 以图表形式显示数据。这种模式有良好的用户界面，可以直观地了解数据，但是当系统中的节点数量过大，监控端会因为需要处理的数据过多而产生较大的时延。加州大学伯克利分校开发的 Ganglia[25]集群监控系统在集群中的每个节点上安装 gmond 程序收集信息，集群管理节点利用 gmetad 收集 gmond 采集的性能数据，与 Supermon 不同的是，集群中的节点之间可以通过组播的形式广播收集到的消息，使消息在集群内共享，Ganglia 的树形结构很适合系统的扩展，具有很高的灵活性。在组播的模式下，当节点规模过大，组播对系统性能会产生一定的影响。这几款监控工具主要侧重于性能指标的监控，对于网络服务监控和故障报警的功能相对缺乏，Nagios[26]的出现正好解决了这些问题。Nagios 是一款功能强大的网络服务和主机监控系统，配置故障条件后利用内部的 4 种故障状态可以实现故障报警，简单的插件设计为 Nagios 提供很高的扩展性，但是 Nagios 配置文件较多，配置步骤烦琐。

Chukwa[27]是一个开源的监控大型分布式系统的数据收集系统，它构建于 HDFS 和 Map/Reduce 框架之上，并继承了 Hadoop 优秀的扩展性和健壮性。在数据分析方面，Chukwa 拥有一套灵活、强大的工具，可用于监控和分析结果来更好地利用所收集的数据结果。图 2-7 为 Chukwa 整体系统架构。

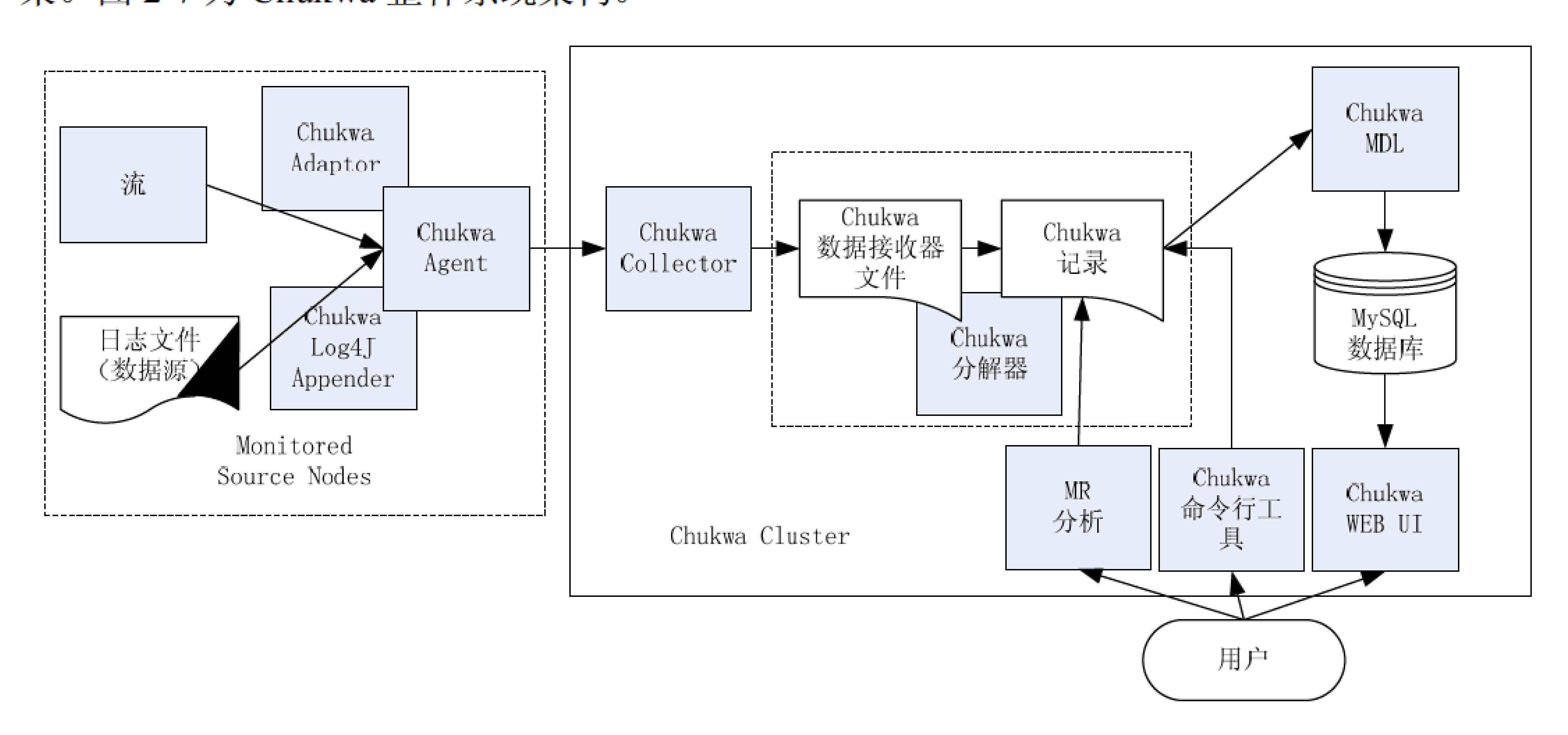


图 2-7　Chukwa 整体系统架构

Chukwa 是 Yahoo 公司开发的 Hadoop 之上的数据采集/分析框架，主要用于日志采集/分析。该框架提供了采集数据的 Agent，由 Agent 采集数据通过 HTTP 发送数据给 Cluster 的 Collector，Collector 把数据搜集进 Hadoop，然后通过定期运行 Map/Reducer 来分析数据，并将结果呈现给用户。

2.4 并行计算与云计算关系

所谓并行计算分为时间上的并行和空间上的并行。

- 时间上的并行：是指流水线技术，这就是并行算法中的时间并行，在同一时间启动两个或两个以上的操作，大大提高计算性能。
- 空间上的并行：是指多个处理机并发地执行计算，即通过网络将两个以上的处理机连接起来，达到同时计算同一个任务的不同部分，或者单个处理机无法解决的大型问题。

空间上的并行导致了两类并行机的产生，按照 Flynn 的说法分为：单指令流多数据流（SIMD）和多指令流多数据流（MIMD）。我们常用的串行机也叫做单指令流单数据流（SISD）。MIMD 类的机器又可分为以下常见的五类：并行向量处理机（PVP）、对称多处理机（SMP）、大规模并行处理机（MPP）、工作站机群（COW）、分布式共享存储处理机（DSM）。表 2-2 为并行计算概述。

表 2-2　并行计算概述表

概述	高性能计算、云端运算	计算机集群	分布式计算	网格计算
方式	Bit-level parallelism	Instruction level parallelism	Data parallelism	任务并行
理论	Speedup	Amdahl 定理	Flynn's taxonomy	Cost efficiency
	Gustafson 定理	Karp-Flatt metric		
元素	进程	线程	Fiber	PRAM 模型
协调	多处理	多执行线程	超执行线程	内存一致性
	Cache coherency	Barrier	同步化	Application checkpointing
编程	Programming model	Implicit parallelism	Explicit parallelism	
硬件	贝奥武夫机群	对称多处理机	Asymmetric multiprocessing	Simultaneous multithreading
	非均匀访存模型	Cache only memory architecture	共享内存	Distributed memory
	Distributed shared memory	超纯量	向量处理机	超级计算机
	Stream processing	通用图形处理器 (GPGPU)		
APIs	POSIX Threads	OpenMP	信息传递接口 (MPI)	Intel Threading Building Blocks
问题	Embarrassingly parallel	Grand Challenge	Software lockout	可扩放性
	竞争危害	死锁	确定性算法	

并行计算科学中主要研究的是空间上的并行问题。从程序和算法设计人员的角度来看，并行计算又可分为数据并行和任务并行。一般来说，因为数据并行主要是将一个大任务化解成相同的各个子任务，比任务并行要容易处理。

并行计算机有以下 5 种访存模型：

（1）均匀访存模型（UMA）。

（2）非均匀访存模型（NUMA）。
（3）全高速缓存访存模型（COMA）。
（4）一致性高速缓存非均匀存储访问模型（CC-NUMA）。
（5）非远程存储访问模型（NORMA）。

2.4.1 并行计算与云计算

云计算是在并行计算之后产生的概念，是由并行计算发展而来， 两者在很多方面有着共性。但并行计算不等于云计算，云计算也不等同并行计算。两者区别如下：

1. 云计算萌芽于并行计算

云计算的萌芽应该从计算机的并行化开始，并行机的出现是人们不满足于 CPU 摩尔定律的增长速度，希望把多个计算机并联起来，从而获得更快的计算速度。这是一种很简单也很朴素的实现高速计算的方法，这种方法后来被证明是相当成功的。

2. 并行计算、网格计算只用于特定的科学领域，专业的用户

并行计算、网格计算的提出主要是为了满足科学和技术领域的专业需要，其应用领域也基本限于科学领域。传统并行计算机的使用是一个相当专业的工作，需要使用者有较高的专业素质，多数是命令行的操作，这是很多专业人士的噩梦，更不用说普通的业余级用户了。

3. 并行计算追求的高性能

在并行计算的时代，人们极力追求的是高速的计算、采用昂贵的服务器，各国不惜代价在计算速度上超越他国，因此，并行计算时代的高性能机群是一个“快速消费品”，世界 TOP500 高性能计算机的排名不断地在刷新，一台大型机群如果在 3 年左右不能得到有效地利用就远远地落后了，巨额投资无法收回。

4. 云计算对于单节点的计算能力要求低

云计算时代并不去追求使用昂贵的服务器，云中心的计算力和存储力可随着需要逐步增加，云计算的基础架构支持这一动态增加的方式，高性能计算将在云计算时代成为“耐用消费品”。

云计算是一种理念，它实际上是分布式技术+服务化技术+资源隔离和管理技术（虚拟化）；把 IT 资源、数据、应用作为服务通过网络提供给用户；把大量的高度虚拟化的资源管理起来，组成一个大的资源池，用来统一提供服务；以公开的标准和服务为基础，以互联网为中心，提供安全 、快速、便捷的数据存储和网络计算服务。图 2-8 为云计算示意图。

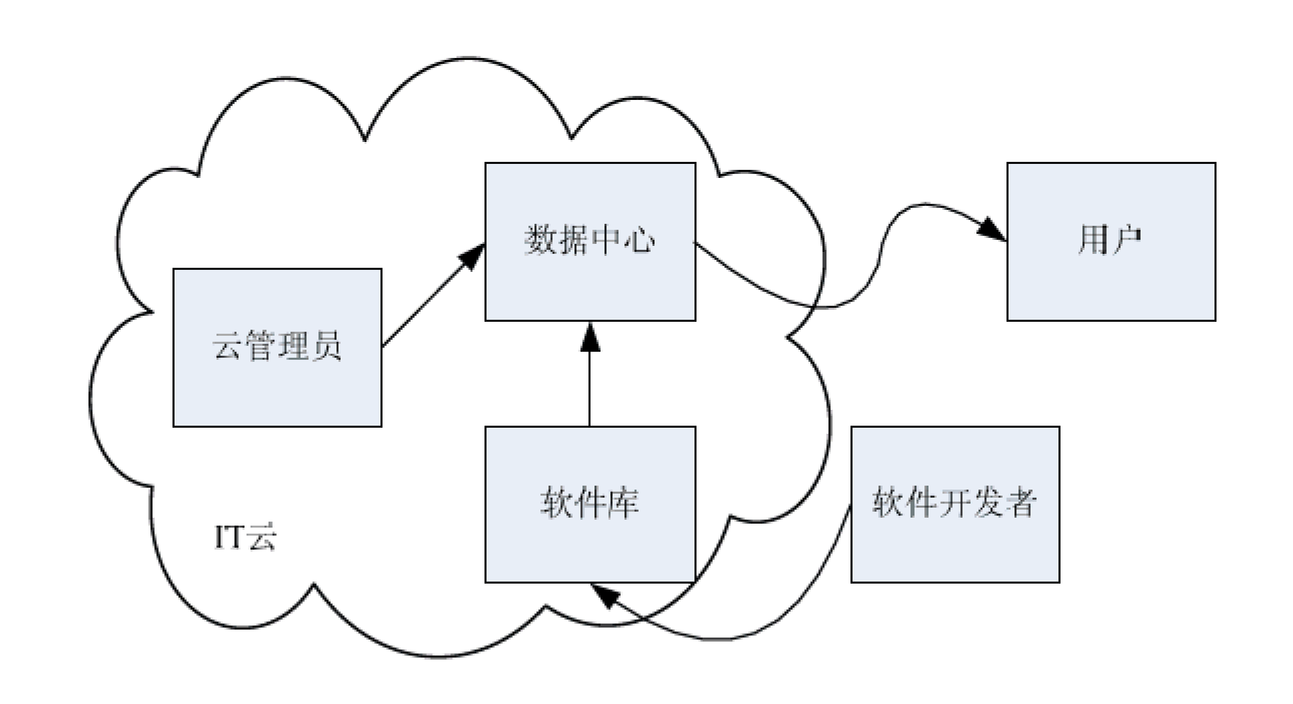


图 2-8　云计算示意图

云计算往往是图 2-8 这个架构图包含的内容，开发者利用云 API 开发应用，然后上传到云上托管，并提供给用户使用，而不关心云背后的运维和管理，以及机器资源分配等问题。

2.4.2　MapReduce

MapReduce 是一种编程模型，用于大规模数据集（大于 1TB）的并行运算。概念“Map（映射）”和“Reduce（归约）”，是它们的主要思想，都是从函数式编程语言及矢量编程语言里借来的特性。它极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。 当前的软件实现是指定一个 Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的 Reduce（归约）函数，用来保证所有映射的键值对中的每一个共享相同的键组[28]。

MapReduce 分布可靠，MapReduce 通过把对数据集的大规模操作分发给网络上的每个节点实现可靠性；每个节点会周期性地返回它所完成的工作和最新的状态。如果一个节点保持沉默超过一个预设的时间间隔，主节点（类同 Google File System 中的主服务器）记录下这个节点状态为死亡，并把分配给这个节点的数据发到别的节点。每个操作使用命名文件的原子操作以确保不会发生并行线程间的冲突；当文件被改名的时候，系统可能会把他们复制到任务名以外的另一个名字上去[29]。

在 Google，MapReduce 用在非常广泛的应用程序中，包括“分布 grep、分布排序、Web 连接图反转、每台机器的词矢量、Web 访问日志分析、大规模的算法图形处理、反向索引构建、文档聚类、数据挖掘、机器学习、文字处理、统计机器翻译以及众多其他领域。”值得注意的是，MapReduce 实现以后，它被用来重新生成 Google 的整个索引，并取代老的 ad hoc 程序去更新索引。MapReduce 会生成大量的临时文件，为了提高效率，它利用 Google 文件系统来管理和访问这些文件。

Nutch 项目开发了一个实验性的 MapReduce 的实现，也即是后来大名鼎鼎的 Hadoop[30]。

Phoenix 是斯坦福大学开发的基于多核/多处理器、共享内存的 MapReduce 实现[31]。

MapReduce 提供了以下的主要功能：

1. 数据划分和计算任务调度

系统自动将一个作业（Job）待处理的大数据划分为很多个数据块，每个数据块对应于一

个计算任务（Task），并自动调度计算节点来处理相应的数据块。作业和任务调度功能主要负责分配和调度计算节点（Map 节点或 Reduce 节点），同时负责监控这些节点的执行状态，并负责 Map 节点执行的同步控制。

2. 数据/代码互定位

为了减少数据通信，一个基本原则是本地化数据处理，即一个计算节点尽可能处理其本地磁盘上所分布存储的数据，这实现了代码向数据的迁移；当无法进行这种本地化数据处理时，再寻找其他可用节点并将数据从网络上传送给该节点（数据向代码迁移），但将尽可能从数据所在的本地机架上寻找可用节点以减少通信延迟。

3. 系统优化

为了减少数据通信开销，中间结果数据进入 Reduce 节点前会进行一定的合并处理：一个 Reduce 节点所处理的数据可能会来自多个 Map 节点，为了避免 Reduce 计算阶段发生数据相关性，Map 节点输出的中间结果需使用一定的策略进行适当的划分处理，保证相关性数据发送到同一个 Reduce 节点；此外，系统还进行一些计算性能优化处理，如对最慢的计算任务采用多备份执行、选最快完成者作为结果。

4. 出错检测和恢复

在低端商用服务器构成的大规模 MapReduce 计算集群中，节点硬件（主机、磁盘、内存等）出错和软件出错是常态，因此 MapReduce 需要能检测并隔离出错节点，并调度分配新的节点接管出错节点的计算任务。同时，系统还将维护数据存储的可靠性，用多备份冗余存储机制提高数据存储的可靠性，并能及时检测和恢复出错的数据。

MapReduce 伪代码

Map 函数和 Reduce 函数是交给用户实现的，这两个函数定义了任务本身。

Map 函数：接受一个键值对（key-value pair），产生一组中间键值对。MapReduce 框架会将 map 函数产生的中间键值对里键相同的值传递给一个 reduce 函数。

```
ClassMapper
methodmap(String input_key, String input_value):
// input_key: text document name
// input_value: document contents
for eachword w ininput_value:
EmitIntermediate(w, "1");
```

Reduce 函数：接受一个键，以及相关的一组值，将这组值进行合并产生一组规模更小的值（通常只有一个或零个值）。

```
ClassReducer
method reduce(String output_key,Iterator intermediate_values):
// output_key: a word
```

```
// output_values: a list of counts
intresult = 0;
for each v in intermediate_values:
result += ParseInt(v);
Emit(AsString(result));
```

MapReduce 提供了一种抽象机制将程序员与系统层细节隔离开来，程序员仅需描述需要计算什么（What to compute），而具体怎么去计算（How to compute）就交由系统的执行框架处理，这样程序员可从系统层细节中解放出来，而致力于其应用本身计算问题的算法设计。

图 2-9 为 MapReduce 工作原理执行流程。一切都是从最上方的 user program 开始的，user program 链接了 MapReduce 库，实现了最基本的 Map 函数和 Reduce 函数。

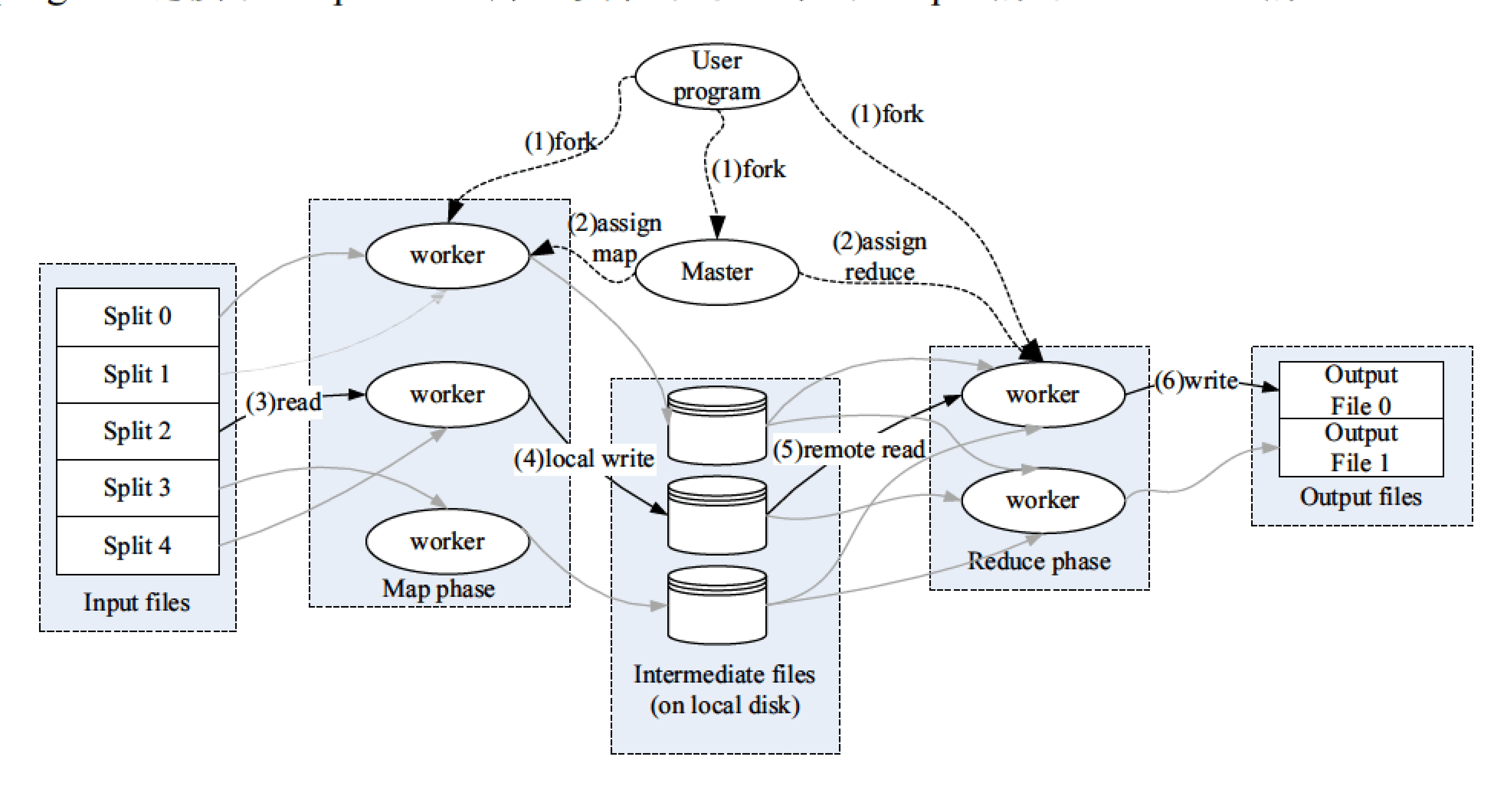


图 2-9　MapReduce 工作原理执行流程

函数说明 pid_t fork(void)：一个现有进程可以调用 fork 函数创建一个新进程。由 fork 创建的新进程被称为子进程。fork 函数被调用一次但返回两次。两次返回的唯一区别是子进程中返回 0 值而父进程中返回子进程 ID。子进程是父进程的副本，它将获得父进程数据空间、堆、栈等资源的副本。注意，子进程持有的是上述存储空间的“副本”，这意味着父子进程间不共享这些存储空间。

（1）MapReduce 库先把 user program 的输入文件划分为 M 份（M 为用户定义），每一份通常有 16MB 到 64MB，如图 2-9 左方所示分成了 split0~4（文件块）；然后使用 fork 将用户进程复制到集群内其他机器上。

（2）user program 的副本中有一个称为 Master，其余称为 worker，Master 是负责调度的，为空闲 worker 分配作业（map 作业或 Reduce 作业），worker 数量可由用户指定。

（3）被分配了 Map 作业的 worker，开始读取对应文件块的输入数据，Map 作业数量是由 M 决定的，和 split 一一对应；Map 作业（包含多个 map 函数）从输入数据中抽取出键值对，每一个键值对都作为参数传递给 map 函数，map 函数产生的中间键值对被缓存在内存中。

（4）缓存的中间键值对会被定期写入本地磁盘。主控进程知道 Reduce 的个数，比如 R 个（通常用户指定）。然后主控进程通常选择一个哈希函数作用于键并产生 0~R-1 个桶编号。Map 任务输出的每个键都被哈希起作用，根据哈希结果将 Map 的结果存放到 R 个本地文件中的一个（后来每个文件都会指派一个 Reduce 任务）。

（5）Master 通知分配了 Reduce 作业的 worker 负责的分区在什么位置。当 Reduce worker 把所有它负责的中间键值对都读过来后，先对它们进行排序，使得相同键的键值对聚集在一起。因为不同的键可能会映射到同一个分区也就是同一个 Reduce 作业（谁让分区少呢），所以排序是必需的。

（6）Reduce worker 遍历排序后的中间键值对，对于每个唯一的键，都将键与关联的值传递给 reduce 函数，reduce 函数产生的输出会添加到这个分区的输出文件中。

（7）当所有的 Map 和 Reduce 作业都完成了，Master 唤醒正版的 user program，MapReduce 函数调用返回 user program 的代码。

（8）所有执行完毕后，MapReduce 输出放在了 R 个分区的输出文件中（分别对应一个 Reduce 作业）。用户通常并不需要合并这 R 个文件，而是将其作为输入交给另一个 MapReduce 程序处理。整个过程中，输入数据是来自底层分布式文件系统（GFS）的，中间数据是放在本地文件系统的，最终输出数据是写入底层分布式文件系统（GFS）的。而且我们要注意 Map/Reduce 作业和 map/reduce 函数的区别：Map 作业处理一个输入数据的分片，可能需要调用多次 map 函数来处理每个输入键值对；Reduce 作业处理一个分区的中间键值对，期间要对每个不同的键调用一次 reduce 函数，Reduce 作业最终也对应一个输出文件。

以下是在客户端、JobTracker、TaskTracker 的层次来分析 MapReduce 的工作原理，如图 2-10 所示 MapReduce 作业运行流程。

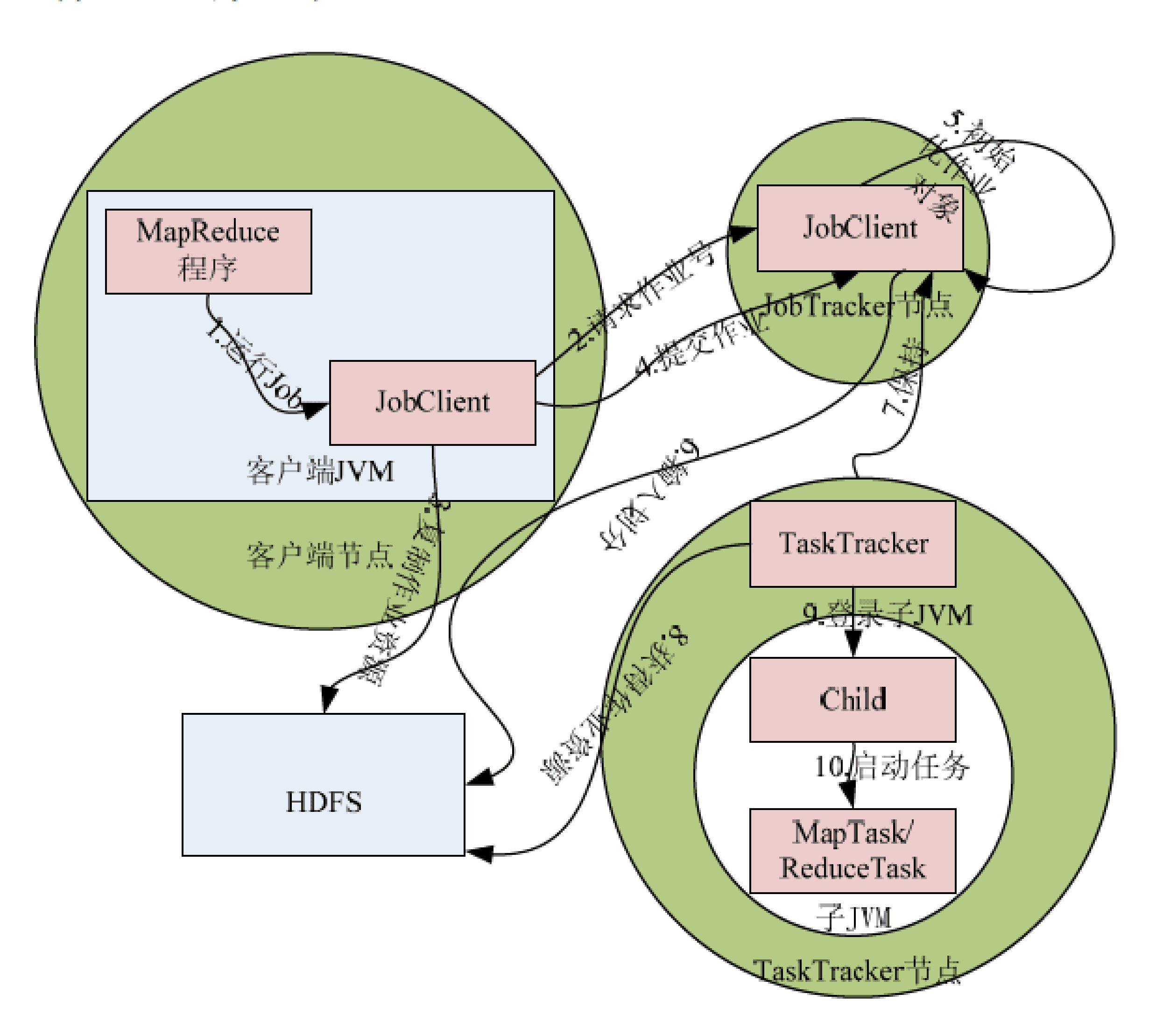


图 2-10　MapReduce 作业运行流程图

图 2-10 MapReduce 作业运行流程分析如下：

（1）在客户端启动一个作业。

（2）向 JobTracker 请求一个 Job ID。

（3）将运行作业所需要的资源文件复制到 HDFS 上，包括 MapReduce 程序打包的 JAR 文件、配置文件和客户端计算所得的输入划分信息。这些文件都存放在 JobTracker 专门为该作业创建的文件夹中。文件夹名为该作业的 Job ID，JAR 文件默认会有 10 个副本（mapred.submit.replication 属性控制），输入划分信息告诉了 JobTracker 应该为这个作业启动多少个 Map 任务等信息。

（4）JobTracker 接收到作业后，将其放在一个作业队列里，等待作业调度器对其进行调度，当作业调度器根据自己的调度算法调度到该作业时，会根据输入划分信息为每个划分创建一个 Map 任务，并将 Map 任务分配给 TaskTracker 执行。对于 Map 和 Reduce 任务，TaskTracker 根据主机核的数量和内存的大小有固定数量的 Map 槽和 Reduce 槽。这里需要强调的是：Map 任务不是随随便便地分配给某个 TaskTracker 的，这里有个概念叫：数据本地化（Data-Local）。意思是：将 Map 任务分配给含有该 Map 处理的数据块的 TaskTracker 上，同时将程序 JAR 包复制到该 TaskTracker 上来运行，这叫“运算移动，数据不移动”。而分配 Reduce 任务时并不考虑数据本地化。

（5）TaskTracker 每隔一段时间会给 JobTracker 发送一个心跳，告诉 JobTracker 它依然在运行，同时心跳中还携带着很多的信息，比如当前 Map 任务完成的进度等信息。当 JobTracker 收到作业的最后一个任务完成信息时，便把该作业设置成“成功”。当 JobClient 查询状态时，它将得知任务已完成，便显示一条消息给用户。

Map、Reduce 任务中 Shuffle 和排序的过程

shuffle 和排序是发生在作业执行阶段，具体说是发生在将 map 输出到 reduce 输入之间的过程。shuffle 主要包括复制、排序、reduce 处理等阶段。shuffle 属于不断被优化和改进的代码库的一部分，从许多方面来看，shuffle 是 Mapreduce 的“心脏”，是奇迹发生的地方。图 2-11 为 Map、Reduce 任务中 Shuffle 和排序的过程。

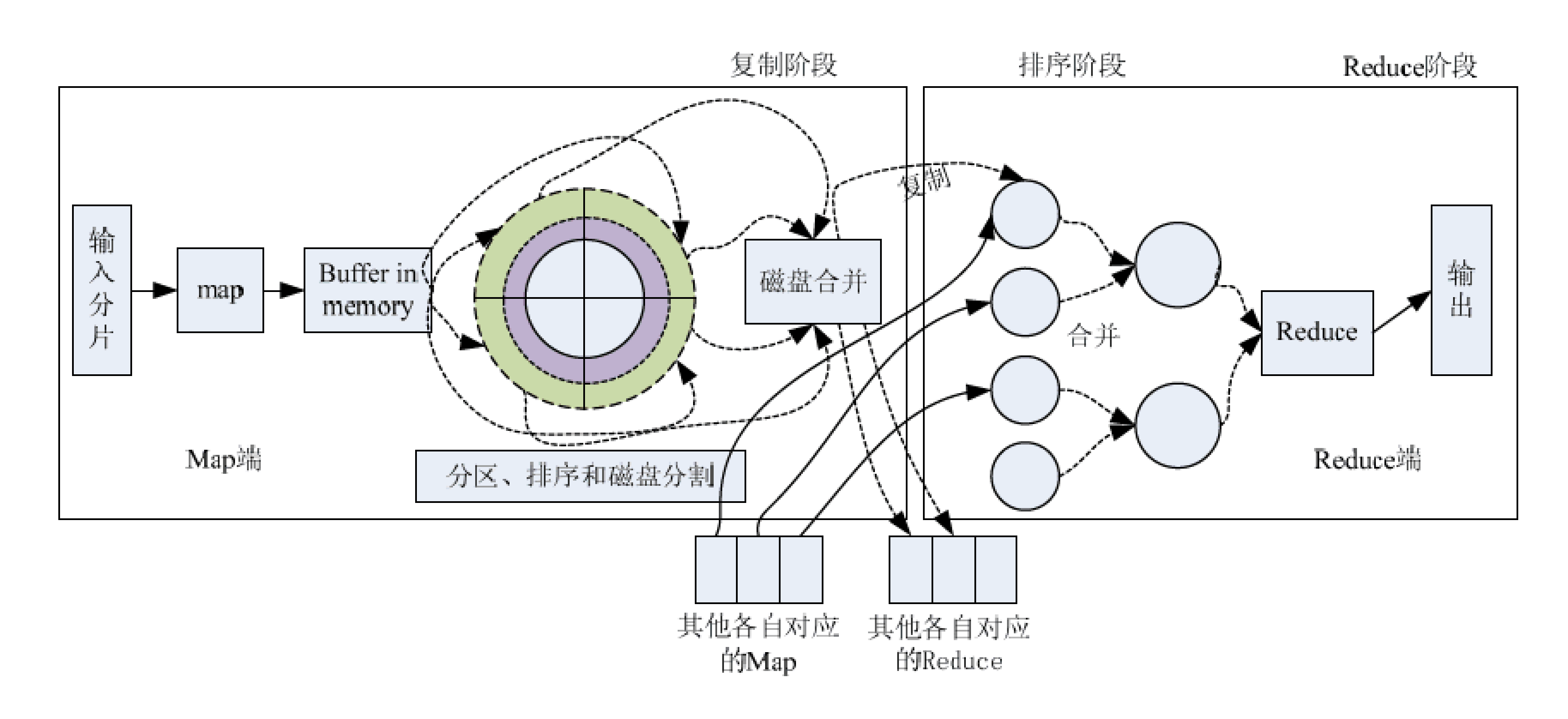


图 2-11　Map、Reduce 任务中 Shuffle 和排序的过程

Map 输出端（分区、排序）

由于 MapReduce 确保每个 Reducer 的输入都按键排序。map 函数开始产生输出时，并不

是简单地将它写到磁盘，而是先利用缓冲的方式写到内存，在内存里对键进行排序。这期间的过程：

（1）线程首先将要传送的 Reduce 的数据划分成与相应的分区对应。

（2）在每个分区中，后台线程按键进行内排序。

（3）如果有一个 combiner，他会在排序后的输出上运行。如果果真制定 combine，则 combiner 就会在输出文件写磁盘之前运行。运行 combiner 的意义在于使 Map 输出更紧凑，这样写本地磁盘和传给 Reducer 的数据更少。

（4）写磁盘时压缩 Map 输出往往是个好主意，因为这样会让写磁盘的速度更快，节约磁盘空间，并且减少传给 Map 的数据量。默认情况下，输出是不压缩的，但是要将 mapred.compress.map.output 设置为 true，就可以轻松地启用此功能。使用的压缩库由 mapred.map.output.compression.codec 指定。

（5）reducer 通过 HTTP 方式得到磁盘上排序后的输出文件的分区。

Reduce 输入端（复制、合并、Reduce 阶段）

（1）每个 Map 完成的时间不同，因此一个 Map 任务完成，Reduce 就开始复制其输出，这就是复制阶段。Reduce 任务有少量复制线程，因此能够并行取得 Map 输出。默认值是 5 个线程，但这个默认值可以通过 mapred.reduce.parallel.copies 属性来改变。Reducer 中的一个线程定期询问 jobtracker 以便获取 Map 输出的位置。如果 Map 输出相当小，则会被复制到 Reduce tasktracker 的内存。否则，被缓冲式地复制到磁盘。

（2）随着磁盘上的副本增多，后台线程会将他们合并为更大的、排好序的文件，这就是合并阶段。这会为后面的合并节省一点时间。注意：为了合并，压缩的 Map 输出都会在内存中解压缩。

（3）Reduce 阶段：对已排序输出中的每个键都要调用 Reduce 函数。此阶段的输出直接写到输出文件系统 HDFS。

图 2-11 Map、Reduce 任务中 Shuffle 和排序的过程流程分析如下。

Map 端

（1）每个输入分片会让一个 map 任务来处理，默认情况下，以 HDFS 的一个块的大小（默认为 64MB）为一个分片，我们也可以设置块的大小 Map 输出的结果会暂且放在一个环形内存缓冲区中（该缓冲区的大小默认为 100MB，由 io.sort.mb 属性控制），当该缓冲区快要溢出时（默认为缓冲区大小的 80%，由 io.sort.spill.percent 属性控制），会在本地文件系统中创建一个溢出文件，将该缓冲区中的数据写入这个文件。

（2）在写入磁盘之前，线程首先根据 Reduce 任务的数目将数据划分为相同数目的分区，也就是一个 Reduce 任务对应一个分区的数据。这样做是为了避免有些 Reduce 任务分配到大量数据，而有些 Reduce 任务却分到很少数据，甚至没有分到数据的尴尬局面。其实分区就是对数据进行 Hash 的过程。然后对每个分区中的数据进行排序，如果此时设置了 Combiner，将排序后的结果进行 Combi 操作，这样做的目的是让尽可能少的数据写入到磁盘。

（3）当 Map 任务输出最后一个记录时，可能会有很多的溢出文件，这时需要将这些文件合并。合并的过程中会不断地进行排序和 Combi 操作，目的有两个：一是尽量减少每次写

入磁盘的数据量；二是尽量减少下一复制阶段网络传输的数据量。最后合并成了一个已分区且已排序的文件。为了减少网络传输的数据量，这里可以将数据压缩，只需要将 mapred.compress.map.out 设置为 true。

（4）将分区中的数据复制给相对应的 Reduce 任务。有人可能会问：分区中的数据怎么知道它对应的 Reduce 是哪个呢？其实 Map 任务一直和其父 TaskTracker 保持联系，而 TaskTracker 又一直和 JobTracker 保持心跳。所以 JobTracker 中保存了整个集群中的宏观信息。只要 Reduce 任务向 JobTracker 获取对应的 Map 输出位置。

那到底什么是 Shuffle 呢？Shuffle 的中文意思是“洗牌”，如果我们这样看：一个 Map 产生的数据，结果通过 HASH 过程分区却分配给了不同的 Reduce 任务，是不是一个对数据洗牌的过程呢？

Reduce 端

（1）Reduce 会接收到不同 Map 任务传来的数据，并且每个 Map 传来的数据都是有序的。如果 Reduce 端接受的数据量相当小，则直接存储在内存中（缓冲区大小由 mapred.job.shuffle.input.buffer.percent 属性控制，表示用作此用途的堆空间的百分比），如果数据量超过了该缓冲区大小的一定比例（由 mapred.job.shuffle.merge.percent 决定），则对数据合并后溢写到磁盘中。

（2）随着溢写文件的增多，后台线程会将它们合并成一个更大的有序的文件，这样做是为了给后面的合并节省时间。其实不管在 Map 端还是 Reduce 端，MapReduce 都是反复地执行排序，合并操作，现在终于明白了有些人为什么会说：排序是 Hadoop 的灵魂。

（3）合并的过程中会产生许多的中间文件（写入磁盘了），但 MapReduce 会让写入磁盘的数据尽可能地少，并且最后一次合并的结果并没有写入磁盘，而是直接输入到 Reduce 函数。

2.5 云计算发展优势

美国市场研究公司评选出 2011 年对多数组织最具战略意义的十大技术和趋势，云计算居首。可见，云计算正是最符合条件的战略技术，是时代发展的必然趋势。戴尔数据中心解决方案部门在最近戴尔迈向云计算之旅的演讲中形容云计算的热度时这样说：“有人认为云计算无所不能，甚至能够解决全球的饥饿问题。”毫无疑问，云计算已经成为 IT 业的主旋律：无论是亚马逊、Google，还是 IBM、微软几乎都异口同声地将“云”定为未来的发展重心。

2010 年可以称为“中国的云计算落地之年”，2011 年云计算在各个行业的各种应用中已经开始崭露头角。各种崭新的云计算应用概念也被提出来，比如智慧城市、虚拟化、公共云、私有云，云存储。特别是伴随着阿里云、百度云、八百客、用友伟库等云计算厂商的兴起，更是让云计算变得炙手可热。

近年来，对于打造高度可扩展的应用程序，软件架构师们挖掘了若干相关理念，并以最佳实践的方式加以实施。在今天的“信息时代”，这些理念更加适用于不断增长的数据集、不可预知的流量模式，以及快速响应时间的需求。

1. 云计算的商业优势

- 前期基础设施投资几乎为零。
- 基础设施即时性。
- 更有效地利用资源。
- 根据使用计算成本。
- 缩短产品上市时间。

2. 云计算的技术优势

- 自动化："脚本化的基础设施"，可以通过充分利用可编程（API 驱动的）基础设施，可重用构建和部署系统;
- 自动扩展：无须任何人工干预，就可以根据需求对应用进行双向扩展。自动缩放提高了自动化程度从而更加高效。
- 主动扩展：基于需求预期和流量模式的合理规划，可以对应用进行双向扩展，从而保持低成本运营。
- 更有效的开发周期：可以很容易地克隆开发和测试环境到生产系统。不同阶段的环境可以很容易地推广到生产系统。
- 改进的可测性：不需要进行硬件耗尽的测试。注入和自动化测试能够持续在开发过程的每一个阶段。我们可以建立一个预配置环境——"即时测试实验室"，仅用于一段时间的测试。
- 灾难恢复和业务连续性：云服务为维护一系列 DR 服务器和数据存储提高了低成本选择。使用云服务，你可以在几分钟内完成将某一地点的环境复制到其他地域的云环境中。
- 流量溢出到云环境：通过几次点击和有效的负载均衡策略，可以创建路由，将超出的访问流量转移到云环境中的一个完整的防溢应用程序。

云计算在设计上提供了概念上的无限可扩展。但是，如果你的架构部署是可扩展的，也无法使用到云计算的可扩展性带来的优势。你必须确定架构中的瓶颈和单点组件，确定架构中哪些是不能按需部署的部分，然后重构应用来调整为可扩展的架构，从而获得云计算的益处。在架构设计时要铭记于心，基础设施和应用架构要协同工作完成可扩展性。图 2-12 解释了一个云应用架构中按需扩展的不同方法。

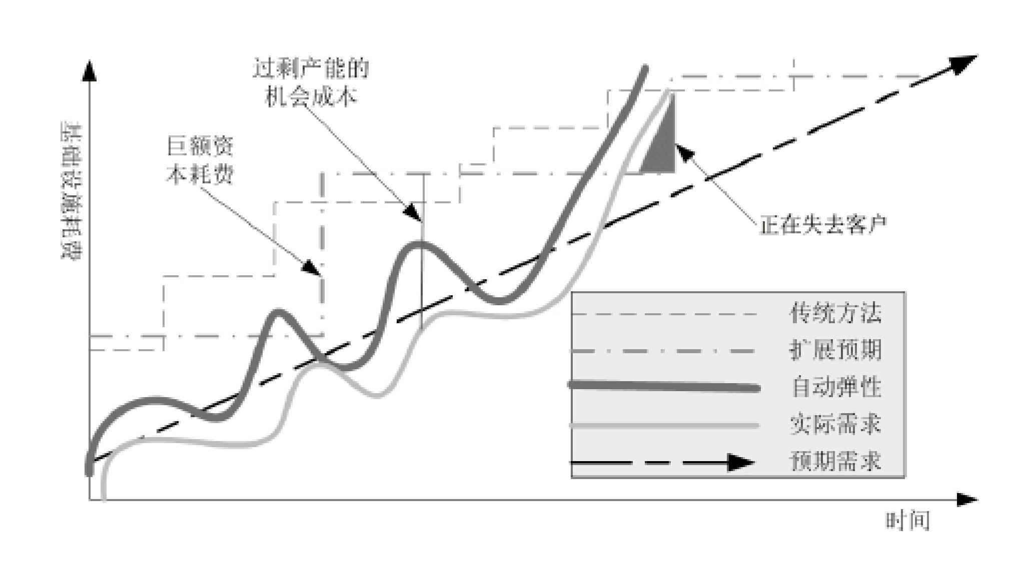


图 2-12　云应用架构中按需扩展的不同方法

- 放大扩展的途径：使用可扩展的应用架构不用担心为了满足需求而大规模投资以及购买更强大的服务器（垂直扩展），这种方法通常工作到一个点，但是在新设备部署前就可以降低成本（见图 2-12 中的“巨额资本破费”）或者满足业务增长的需要（见图 2-12 中“正在失去客户”）。
- 传统向外扩展的途径：创建水平扩展的架构和投资小块的基础设施。大多数业务或大规模 Web 应用都采用如下的模式：分布式应用组件、联合数据集和 SOA 的设计。这种方法通常比放大扩展更有效。然而，这需要准确的业务预期才能实现满足需求的部署，经常会导致容量过剩（“烧钱”）和持续人工监测（“浪费人力成本”）。此外，如果遇到业务的爆发式增长，系统将无法正常工作。

传统结构一般要预测几年内系统所需资源的数量，如果预计不足，应用将没有马力处理预期外的流量，从而导致客户的不满。如果预计过高，又造成资源浪费。

按需部署和弹性是云计算的天然方式（自动弹性），使基础设施与真实需求尽量匹配，因而可以提供资源利用率及压缩成本。

2.6 向云实现迁移

由于对灵活、敏捷、高效的基础设施的需求不断增加，企业管理都趋向云计算解决方案。在过去，大多数的 IT 架构和关键任务应用程序被保存在公司内部数据中心内。然而，时至今日状况已经改变，因为很多企业已经或有计划将关键系统迁移到云计算平台。事实上大多数企业现在都明白，实施云计算和迁移的关键基础设施资源的托管环境能带来许多好处。云计算基础设施指的是支持云计算模型计算需求的软件和硬件组合，包括存储单元、服务器、虚拟化监控和网络软件。众多迁移到云计算中的业务已经充分享受到了云计算带来的便利，以下是业务应该转移到云计算的十大理由。

1. Cost-efficient

云计算能够降低硬件的高成本，用户可以根据自己的预算来选择业务模式。它能够保证用户根据自己的需求付费，而不必为未使用的服务买单，并且设置系统也相对便宜。

2. 增进合作（Increased collaboration）

云计算能够使得业务团队的成员在任意地点访问、编辑并共享文档，大大提高生产力和业务支持协作的效率，包括基于云计算的工作流程和文件共享应用程序的实时更新。

3. 无缝集成（Seamless integration）

已经从孤立的应用程序中有所转变，企业业务更喜欢云计算提供的集成的方式，云计算提供了一个将基础设施服务、数据管理、发展等集成到一起的平台。

4. 促进可伸缩性（Promotes Scalability）

每个业务都能够根据经验进行变化——扩大、缩小或者随着季节的变化而变化，云计算

有能力适应并应对这些变化，这可以根据业务的需要促进可伸缩性，用户不再需要在扩大或者缩小规模时不断地改变软件。

5. 提供良好的用户控件（Offers great user-control）

员工可以携带自己的设备办公，不依赖于 IT 部门提供的设备。这需要更高水平的用户控制，其核心是 Active Directory，它是个性化、合规、数据安全的基础，让用户在业务中学习数据洞察力，提高用户价值。

6. 宽松的软件维护（Tension-free software maintenance）

所有的服务器都将是远程的，用户的工作是在虚拟基础设施的环境下展开。当有一个新的软件，云服务提供商将保证软件的升级，这能有效确保系统运行平稳，用户不必在维护软件上花时间和精力——可以把这些时间用在提升关键业务上。

7. Great speed

实现和配置云服务的时间仅仅只需要几个小时，而在迁移到云计算之前同样的服务需要设置本地服务器和设备，周转的时间也很快。更换虚拟桌面和解决虚拟化的问题也不再像过去一样需要几周的时间，往往很快就能解决。

8. 高灵活性（High flexibility）

云计算提供了巨大的灵活性，员工可以在任何地方工作。不用再限制员工的工作站访问信息和工作任务，虚拟桌面基础设施（VDI）使得 IT 环境更加灵活且能够迅速应对变化。此外，随着云计算提供的智能市场的视角，用户可以享受到快速变化的营销活动。这种灵活性使得公司能够在用户最需要的时间提供最正确的服务。

9. 大数据（Big data）

大数据是通过多个来源的信息整理的数据，这些数据通过云计算的分析工具进行分析，可以帮助业务经理学习和了解客户的信息。大数据分析技术可以用来迅速向任何行业提供详细的数据。

10. 安全性能高（High security）

云通常比本地的基础设施更加安全，云能够确保用户的没有授权的数据免受黑客和其他威胁，用户甚至可以在远程删除遗失的笔记本电脑中的数据，保证商业机密的安全。

云计算及其基础设施的成功实现能带来极大的好处，这就是为什么用户需要花时间把业务迁移到云中，在每个阶段监测植入，并确保执行和定期维护。

把基础架构从物理迁移到虚拟才能带来最好的资源利用率，然后再迁移到云，一步一步地进行，轻松入云。首先要分析物理环境。当把物理架构加进环境之后，资本支出就会增加。仔细分析环境能帮企业理清没有得到完全利用的资产。分析完成后，物理机到虚拟机的迁移就可以有效提升资源利用率，免除了对新物理架构的需求，减少了管理费用。我们要看一下哪些应用支持虚拟化，以此为依据对应用进行分类。分类标准可以有很多，比如基于平台，或是否需要中间件对应用分类，同样的基于数据库来分类也可行。对环境的测试和评

估，能帮助企业准确发现哪些应用存在不支持虚拟化的可能。企业级应用一般而言都需要高CPU 能量和大数据库，因此不推荐将其转入虚拟化环境。

完成了物理环境分析，下面要做的就是整合并虚拟化服务器。服务器需求一直存在变动，这样使得特定的服务器有时会空闲。在这样的情况下，应该实现整体分析，包括使用模式，确定一下计算容量，然后才可以执行物理机到虚拟机的迁移。在高峰时段或者升级时分析计算需求，这些需求会影响性能和管理。此外，还需要将服务器分离和组成。如果有应用在两个数据库运行，就得用中间件服务器或者运行多数据库的 SQL 服务器。整合好架构之后，要对环境进行测试，避免任何网络和存储故障，这一步完成后就可以开始虚拟化。

这一步骤之后，要做到就是网络和存储虚拟化。应该分析网络和存储架构，发现可能的性能问题。针对分离和孤立网络，可以使用虚拟局域网配置，要把产品的流量和其他流量分开，确保适合的带宽利用率。在存储方面，最重要的是可扩展性。容量规划和管理的首要问题就是存储使用模式的分析。IBM、HP 都有测量和报告数据，以便实现更佳性能和容量规划。另外企业应该测试存储，确保能管理 hypervisor 负载，支撑虚拟化。除了这些，企业还得观察自动化存储管理，这样做能让存储资源安排在多租户或者空中架构中，实现在不同应用中共享存储。

现在就可以向云实现迁移。架构向云的迁移也需要有步骤地进行。最初可以少迁移一些关键应用和相关架构。业务关键的架构应该以之前的成功步骤为基础，确保物理产品的环境已经卸下，但不要完全退役。一旦发生任何意外，物理产品环境可以再次利用。物理环境要留着，运行那些不能虚拟化的应用和服务器。应该确保服务供应商符合行业标准，同时严格的服务水平协议（SLA）和规范的报告必不可少，而且建议做好严格的各级访问控制。

2.7 本章小结

云计算（Cloud Computing）是分布式处理（Distributed Computing）、并行处理（Parallel Computing）和网格计算（Grid Computing）的发展，或者说是这些计算机科学概念的商业实现。本章从概念到技术详细介绍了云计算的服务形式、实现机制、编程模式、虚拟化及数据存储等云计算相关的技术，同时详解了并行计算与云计算的关系。最后，阐述了云计算的发展趋势及数据和业务向云迁移的因素与分析。

第 3 章

◂ 大数据与云计算关系 ▸

很多行业受益于“数据中心作为一个枢纽”，越来越多的以云计算为中心的生态系统合作伙伴集中在一个关键的数据中心，如金融交易、网页和在线服务或是媒体内容的企业。众所周知，这些企业有大量的数据需要进行处理和管理。随着移动智能设备的普及，云计算服务和云应用在云平台的支撑下，让这些庞大的数据得以保存和处理，数据的价值不在于多，而是如何挖掘到有价值的数据，这需要借助云服务和云应用的能力。这也是业界将云计算和大数据相提并论的原因所在，到底云计算与大数据是怎么样的关系呢？

云计算已然走下神坛开始步入应用阶段，而大数据的催生反过来体现了云计算的价值所在。很多 IT 技术人员想必已经注意到业界对于新趋势的关注已由原来的云计算转移到大数据上，越来越多的企业开始推广大数据相关的服务和产品，越来越多的企业将企业数据作为企业资产进行管理和变现，已经开始从数据抽象、数据共享和数据估值开始启动大数据战略。对于大数据趋势并不像云计算那样主要集中在概念层面的讨论，主要是在技术层面的研究。企业视大数据为企业的生命、企业的新竞争力，要想在同类行业中脱颖而出赢得市场，大数据的支持是必不可少的，所以企业纷纷制定大数据战略，无论是互联网企业还是传统企业，都在大数据时代不甘示弱，而大数据时代的特性注定了它与云计算的不解之缘。大数据推动云计算的落地，云计算促进大数据的应用。

3.1 云计算与大数据关系

云计算的关键词在于“整合”，无论是通过现在已经很成熟的传统的虚拟机切分型技术，还是通过 google 后来所使用的海量节点聚合型技术，它都是通过将海量的服务器资源通过网络进行整合，调度分配给用户，从而解决用户因为存储计算资源不足所带来的问题。

大数据正是因为数据的爆发式增长带来一个新的课题内容，如何存储如今互联网时代所产生的海量数据，如何有效地利用分析这些数据等。

他俩之间的关系你可以这样来理解，云计算技术就是一个容器，大数据正是存放在这个容器中的水，大数据是要依靠云计算技术来进行存储和计算的。

关于云计算与大数据直接的关系众说纷纭，云端互通对于云计算和大数据关系做以下三点认识：

首先，云计算与大数据之间是相辅相成、相得益彰的关系。大数据挖掘处理需要云计算作为平台，而大数据涵盖的价值和规律则能够使云计算更好地与行业应用结合并发挥更大的作用。云计算将计算资源作为服务支撑大数据的挖掘，而大数据的发展趋势对实时交互的海量数据查询、分析提供了各自需要的价值信息。

其次，云计算与大数据的结合将可能成为人类认识事物的新的工具。实践证明人类对客观世界的认识是随着技术的进步以及认识世界的工具更新而逐步深入。过去人类首先认识的是事物表面，通过因果关系由表及里，由对个体认识进而找到共性规律。现在将云计算和大数据结合，人们就可以利用高效、低成本的计算资源分析海量数据的相关性，快速找到共性规律，加速人们对于客观世界有关规律的认识。

第三，大数据的信息隐私保护是云计算大数据快速发展和运用的重要前提。没有信息安全也就没有云服务的安全。产业及服务要健康、快速地发展就需要得到用户的信赖，就需要科技界和产业界更加重视云计算的安全问题，更加注意大数据挖掘中的隐私保护问题。从技术层面进行深度的研发，严防和打击病毒与黑客的攻击；同时加快立法的进度，维护良好的信息服务环境。

3.2 大数据与云计算的融合是认识世界的新工具

管好资源的基础是将这些资源真正形成创建、服务能力和高可靠能力。云计算是尽力可为的计算，不保证质量，它是从计算通信平台向计算平台和智能平台转换中出现的一类平台。大数据给技术研究者、产业界带来了很多机会，在当前互联网二次价值信息探索之际，管好数据、管好资源是云计算要做的工作。

几年前大家还都在谈论什么是云计算，而现在其已成为产业共识，所有的 IT 企业都在使用云，Google、微软、亚马逊、IBM 都在大规模部署云。可以说云计算已经真正成为一种技术发展的趋势，而且作为一种重要的部署，正在走向良性发展的通道。

高调的厂商，比如 AWS、Google、微软、IBM 和 Rackspace 等，都提供云基础的 Hadoop 和 NoSQL 数据库平台来支持大数据应用程序。很多初创公司都引入了云平台上的管理服务，按需部署自己的系统。大数据和云计算的融合往往是互联网公司的首选项，尤其是初创的软件和数据服务供应商。

1. 初识大数据云的融合，企业黑白子布局犹豫

很多主流公司并不像互联网公司那样看重云端数据管理。一些公司担心云端的数据安全和隐私保护；一些公司还在大型机和其他本地系统里运行大部分操作，存储在本地的数据量之大，让数据迁移充满挑战。另外，现存数据中心可用的处理能力让 AWS 和 Google 等公有云的成本优势不值一提，即使公司对于云系统所谓的降低成本、增加弹性有兴趣，最终也未必会选择它。以花旗集团为例，随着网络成为普及的应用界面，金融服务公司面对的是洪水

般的非结构化数据，它还需要处理线上金融应用程序中不同的数据结构。这些挑战让花旗集团最后选择了 MongoDB NoSQL 数据库。MongoDB 获得了 AWS 和其他云平台的支持。花旗数据公司负责平台工程的全球领导者 Michael Simone 表示，花旗选择了在云端应用该软件。不过它应用的是私有云，应用限定在纽约公司的防火墙内，由它的 IT 部门全权管理。在纽约的 MongoDB 大会上，Simone 告诉与会者："目前，我们还没有扩展私有云或集成公有云的打算。花旗集团的数据中心很大，技术积累也很深厚，我们可以构建自己内部部署的云计算。"

2. 小荷才露尖尖角——大数据云轻盈起步

总体来看，在云端运行大数据系统仍然是小众行为。在数据仓库研究院开发的大数据成熟度模型中，十个月内有 222 名 IT 和业务专家完成了线上测评，只有 19%的人表示它们的组织在用公有云、私有云和混合云支持大数据应用程序，另有 40%的人表示正在考虑云部署，同时有超过三分之一的人表示它们没有使用云计算的计划。在企业管理协会和 9sight 咨询公司开展的线上调查中，云计算使用比例略高：259 名受访者中，39%的人表示他们的大数据安装包括云系统。WeatherChannel 公司是采用了公有云的案例，Basho 技术公司在 AWS 可用性区域的多个分区运行了 Basho 技术公司的 NoSQL 数据库 Riak 的复制实例，处理和存储来自卫星、雷达系统、天气站等来源的混合数据。该数据库每 5 分钟就为预测引擎更新 3.6 万多地理天气网格的视图，它还用于归档历史数据。美国 TWC 公司执行副总裁兼 CIO Bryson Koehler 认为，Riak 的容错技术和同时支持内存和硬盘存储的功能特别好。经过比较，因为处理效果低，主流关系型数据库并不能适应高容量的云环境，至少不能以较低的成本适应高容量的云环境。但是，在云端部署 NoSQL 软件也是旨在扩大 TWC 灵活性的更广泛的 IT 战略的题中之义。公司在 Google 云和 AWS 上运行应用程序，以免被任何供应商或技术锁定。

3. 大数据云使得企业有更多选择、更多可能

公有云供应商已经为了满足大数据需求，扩展了数据管理能力，不止包含关系型数据库。例如，亚马逊近几年拓宽了 AWS 云选项，包含了很多新兴技术，比如 NoSQL 数据库 DynamoDB、Hadoop 部署 ElasticMapReduce 和 ElastiCache 内存缓存服务、Redshift 数据仓库和 Kinesis 流数据系统。美国咨询公司 Cloud Technology Partners 高级副总裁 DavidLinthicum 表示："AWS 和其他云供应商也创建了相当成熟的服务。一些可用的数据管理云平台已经发展到第五代、第六代了。"

4. 需求——大数据云融合的源泉

例如加拿大海洋网络（ONC）是一家非营利性机构，该机构管理着英属哥伦比亚的一对海洋气象台，计划建立一个公司内部私有云，为使用海洋传感器提供数据的应用模拟地震和海啸创造条件，目标在于更加准确地预测可能发生的自然灾害带来的后果，为政府当局提供预防措施，缓解自然灾害给人们带来的影响，Benoit Pirenne 这样说道，他是 ONC 的数字基础设施主管。该机构位于维多利亚大学，2015 年春天得到了一项三年项目的批准和资金支持。计划进行的分析工作包括收集传感器的多次测定结果，运行预测模型以得出可能发生的所有情况集，但是完成这项工作需要大量数据和强大的计算能力。Pirenne 说道："要计算现

实状况中的‘模拟’几乎是不可能完成的任务，就算在非常高级的平行云系统中也不行。”因此，ONC 正在与 IBM 合作构建一个内部云处理流程和分析工作。

新兴的管理服务供应商——例如 Altiscale、BitYota、Qubole、Treasure Data 和 Rackspace'sObjectRocket 附属公司等——他们通过以低于云平台供应商的价格接管部署和管理任务，能够为企业用户将大数据云装置做得更方便、更划算。

美国的 Sellpoints 公司是一个线上营销和分析服务供应商，使用 Hadoop 和 Spark 的流程工具迅速构建查询表格，查询数据量达到 TB 用户网页活跃度数据。

管好资源的基础是将这些资源真正形成创建、服务能力和高可靠能力。云计算是尽力可为的计算，不保证质量，它是从计算通信平台向计算平台和智能平台转换中出现的一类平台；大数据给技术研究者、产业界带来了很多机会，在当前互联网二次价值信息探索之际，管好数据、管好资源是云计算要做的工作。

同时，如何用好资源也是非常重要的问题。用好资源是硬币的两个面，分别代表着云计算管理和大数据分析。首先，一个重要方面是资源的共享和管理。我们都知道资源和数据是重要的基础设施，在整个社会信息化发展中占有重要位置；另一方面，资源本身也是一个重要的耗能产业，相关数据分析表明，ICT 是全球第五大耗能产业。

例如在 facebook 应用中，某天终端应用量发生爆炸性增长，从 50 台一下子变成 3000 台；再如去年光棍节淘宝交易额突破新纪录，在应用中产生了动则几百万、上千万的访问量。这就要求我们管好资源，配置好资源，同时保证系统的可靠性也非常重要。在这样的供给短时间极大增长、爆炸式资源需求环境下，如何建立高可靠的资源管理就是我们云计算面临的首要挑战。

3.3 大数据隐私保护是大数据云快速发展和运用的重要前提

大数据、云计算是目前互联网发展的一个重要方向，更是信息和网络安全发展的一个重要方向。用户为了能够方便地管理、存储自己单位或个人的大量敏感信息，往往将这些信息寄存在一些专营计算公司的“存储云片”上，而这些所谓的“云片”在什么地方、掌握在何人手里，用户实际上根本不知道。明眼人一看就知道，这无异于玩火、无异于将自己的敏感信息送给人家（敌方或对手）、无异于往空中抛钱、无异于“此地无银 300 两”，因为所谓的“云片”，其实就是一个信息的“小件、大件寄存器”，存储在这些“云片”上的寄存器，毫无安全性和保密性而言。

没有网络的安全就没有数据的安全，没有数据的安全就没有信息安全，没有信息安全也就没有云服务的安全，归根到底，没有网络的安全就没有国家的安全。产业及服务要健康、快速地发展，就需要得到用户的信赖，就需要科技界和产业界更加重视云计算的安全问题，更加注意大数据挖掘中的隐私保护问题。同时加快立法的进度，依法保护信息安全，维护良好的信息服务的环境。

3.3.1 云计算的安全隐私

云计算的出现给信息技术领域带来了重大变革，为用户提供了很好的服务，但是这种变革也给信息安全和网络安全带来了很大的冲击，云计算的安全问题越来越成为云计算各界关注的重点。由于所涉及的资源由多个管理者所有，各管理者之间存在利益问题、信任问题，面对各种各样的冲突，我们无法提供统一规则来部署安全防护措施。另外，由于云计算没有统一的基础设施、没有统一的用户管理要求、没有统一的安全边界，所以对用户数据的安全与隐私保护是极大的威胁。

云计算的安全问题从数据存储方面来讲，主要包括用户放在云平台的数据是否安全，例如是否被修改、被泄露、被损毁；从计算方面来讲，主要指用户在云平台的操作是否会被监视、是否会被重演、是否会被篡改等；从云平台的生态系统来讲，主要说的是云平台本身是否可靠、云平台提供的服务是否正确、云平台及云服务提供商是否乱收费等。云计算的这些安全问题，主要来自于两个方面：一方面云服务提供商是不诚实的、不可靠的，它们随意窃取用户的信息或者随意使用用户部署在云平台的应用程序，我们可以通过采用独立于云服务提供商的第三方来进行有效防范；关键的另一方面，云平台可能会受到外部的攻击或者侵入，通常我们会通过云服务提供商备有各种访问控制和身份认证以及数据加密的机制来予以防护，或者云服务提供商通过基于角色的访问控制和基于联邦的身份管理来进行防护。

3.3.2 大数据的安全隐私

在大数据环境下，各行各业的安全需求正在发生改变，从数据采集、数据整合、数据存储、数据分析、数据挖掘再到数据发布，这一流程已经成为新的完整链条。随着数据量的增大和集中，整个链条中数据的安全威胁也越来越多，数据的安全隐私保护也越来越困难，大数据的安全隐私问题已经成为各企业关注的重点。另外，大数据的发展对个人造成了用户隐私与便利性的冲突：消费者得益于大数据技术，会以更低的价格买到更符合自己需求的商品；但同时，随着个人购买偏好、健康和财务状况的海量数据被收集，人们的隐私也被破坏。整体来说，目前的大数据主要存在基础设施安全、存储安全、网络安全、隐私安全四方面的安全问题。

隐私性指的是一些数据的信息保护。在云计算的大环境下，隐私性的定义比传统意义上更加复杂，它不仅仅指数据的内容本身，还包括数据的结构信息、用户的访问模式、访问历史等。对于云端数据的检索来说，需要保护用户的搜索内容的隐私性，数据拥有者建立在云端索引的隐私性；对于云端数据的存储来说，要保证用户数据在云端的隐私性，在存储到云端后要保证数据是完整的，当数据信息不完整时能够及时恢复；对于云端数据的计算来说，既要保证计算结果的安全隐私，又要保证计算内容的安全隐私[33]。整体来说，由于云计算环境多租户以及开放性等原因，使得云端数据安全隐私性的保护在变得异常困难，但又非常重要。

当前，云端数据安全隐私服务主要提供两种模式：一是用户将自己本地的数据存储至云服务商处，云服务商为其提供数据的存储、管理等服务，同时提供给经授权的其他合法用户检索下载；二是用户可以使用云服务商提供的强大计算资源对数据进行计算，以得到相应结果，从而节省了用户自己的计算资源。图 3-1 为阿里巴巴大数据云技术层次图，图 3-2 为中

国移动大数据云平台。

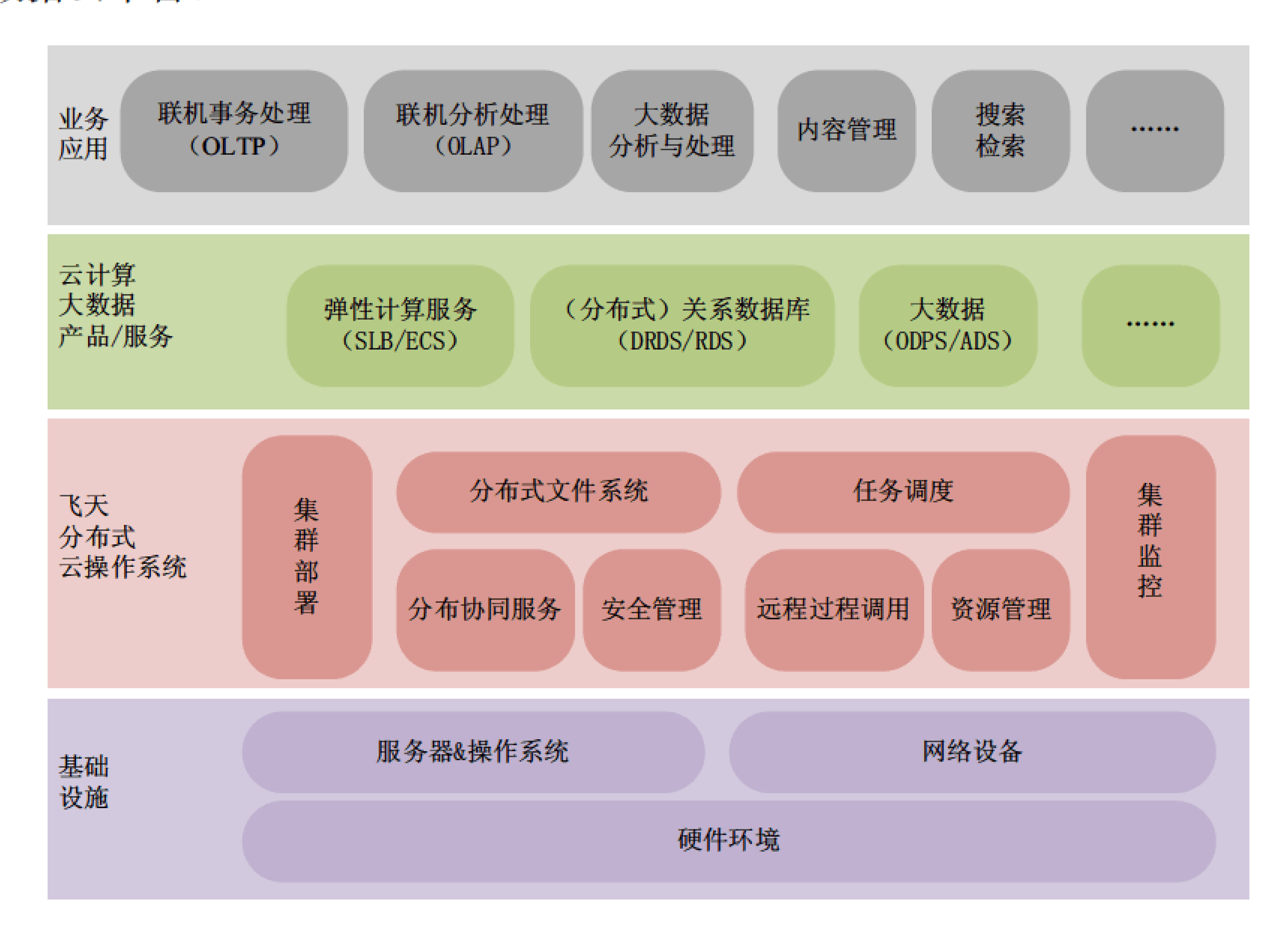


图 3-1　阿里巴巴大数据云技术层次图

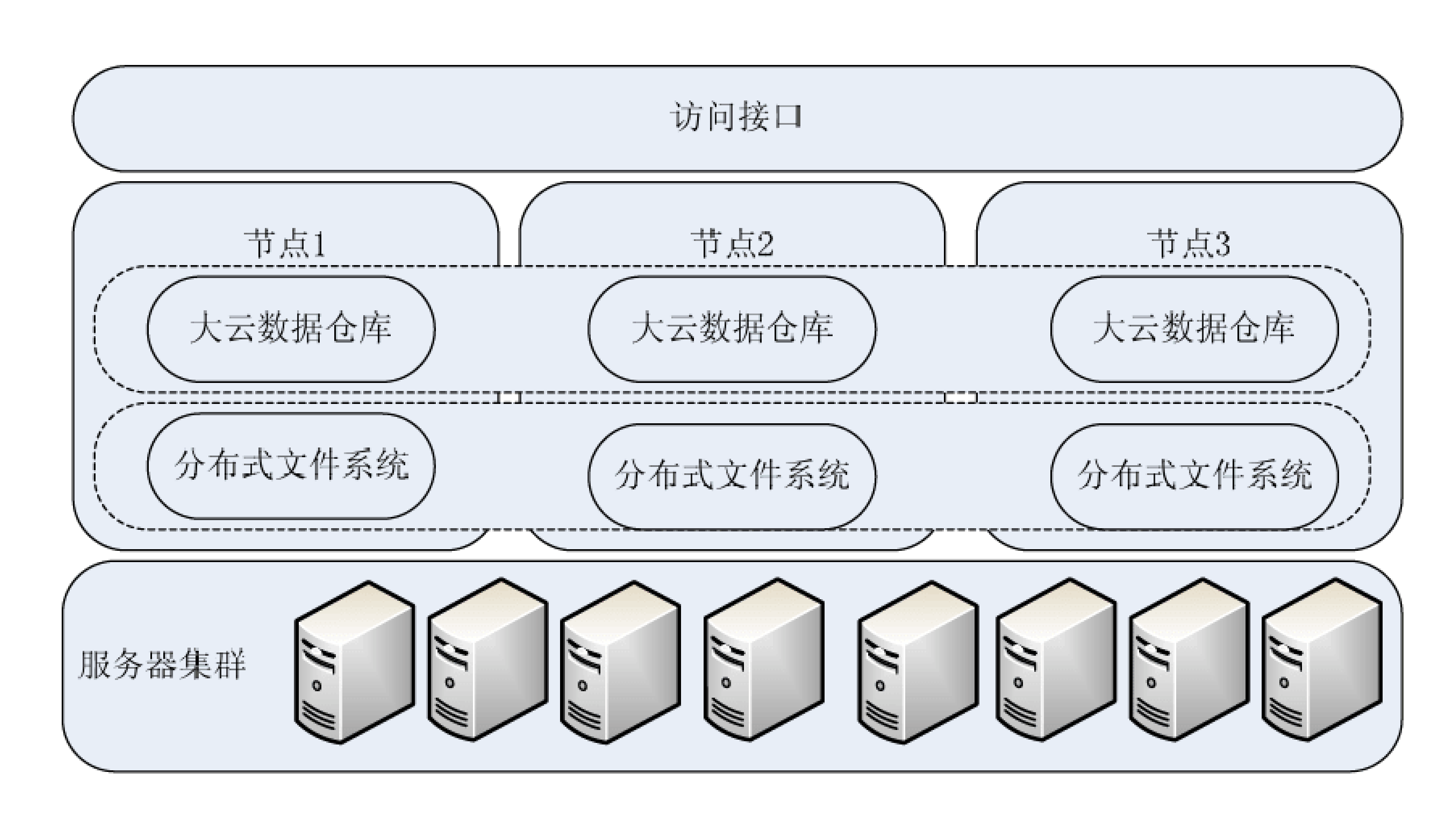


图 3-2　中国移动大数据云平台

云计算和大数据的安全隐私保护是云端大数据平台这一关键技术快速发展和运用的重要前提。云计算、大数据的产业及相应提供的服务要健康、快速的发展，就需要从用户层面重视云平台下的大数据安全隐私保护的问题。从技术层面进行深度研发相关的技术，严防和打击计算机病毒和不法黑客的攻击，确保数据信息拥有者的自主权。从科研层面，我们要研究云端大数据平台的安全隐私保护方面的算法及协议，并运用到具体的实践中去。云计算平台下的大数据安全隐私保护问题，是社会各界必须要高度重视的问题，良好的云端大数据平台的安全隐私保护是保证云计算、大数据快速发展和运用的重要前提。

目前，云服务提供商对用户所能够提供的在线计算和隐私性保护还非常有限。从传统的

加密算法来讲，为了保护自己数据的隐私性，数据拥有者会对上传的数据进行加密，只要加密算法的选择恰当，密钥的保护完好，我们就认为数据的隐私性是安全的。但是在云计算的服务当中，还必须要求对一些计算的相关信息进行保护[34]。还可以基于云端多服务器的安全计算服务解决数据的隐私性，由多个云服务器（Cloud Server）负责存储经过加密处理的数据集，再由一个代理服务器（Proxy）负责按照用户的要求对相关数据进行计算，生成一个加密结果，返还用户之后，最终由用户解密，生成想要的结果。这种方案节省了数据拥有者在本地的存储空间及数据管理成本，同时，利用云计算的功能获得了更快捷、更准确的计算能力，有效地保护了用户数据的隐私性及相关计算请求的隐私性。

3.4 大数据成就云计算价值

当大数据遭遇云计算，从技术上看，大数据与云计算的关系就像一枚硬币的正反面一样密不可分。大数据必然无法用单台的计算机进行处理，必须采用分布式计算架构。它的特色在于对海量数据的挖掘，但它必须依托云计算的分布式处理、分布式数据库、云存储和虚拟化技术。

通常情况下，我们容易将大数据与云计算混淆在一起。著名的麦肯锡全球研究所给出大数据定义是一种规模大到在获取、存储、管理、分析方面大大超出了传统数据库软件工具能力范围的数据集合，具有海量的数据规模、快速的数据流转、多样的数据类型和价值密度低四大特征。对于云计算，则是一种基于互联网的计算方式，通过这种方式，共享的软硬件资源和信息可以按需求提供给计算机和其他设备。借用大数据云计算关系一文中的直白介绍就是云计算是硬件资源的虚拟化，而大数据是海量数据的高效处理。从结果来分析，云计算注重资源分配，大数据注重的是资源处理。一定程度上讲，大数据需要云计算支撑，云计算为大数据处理提供平台。图 3-3 为大数据与云计算关系。

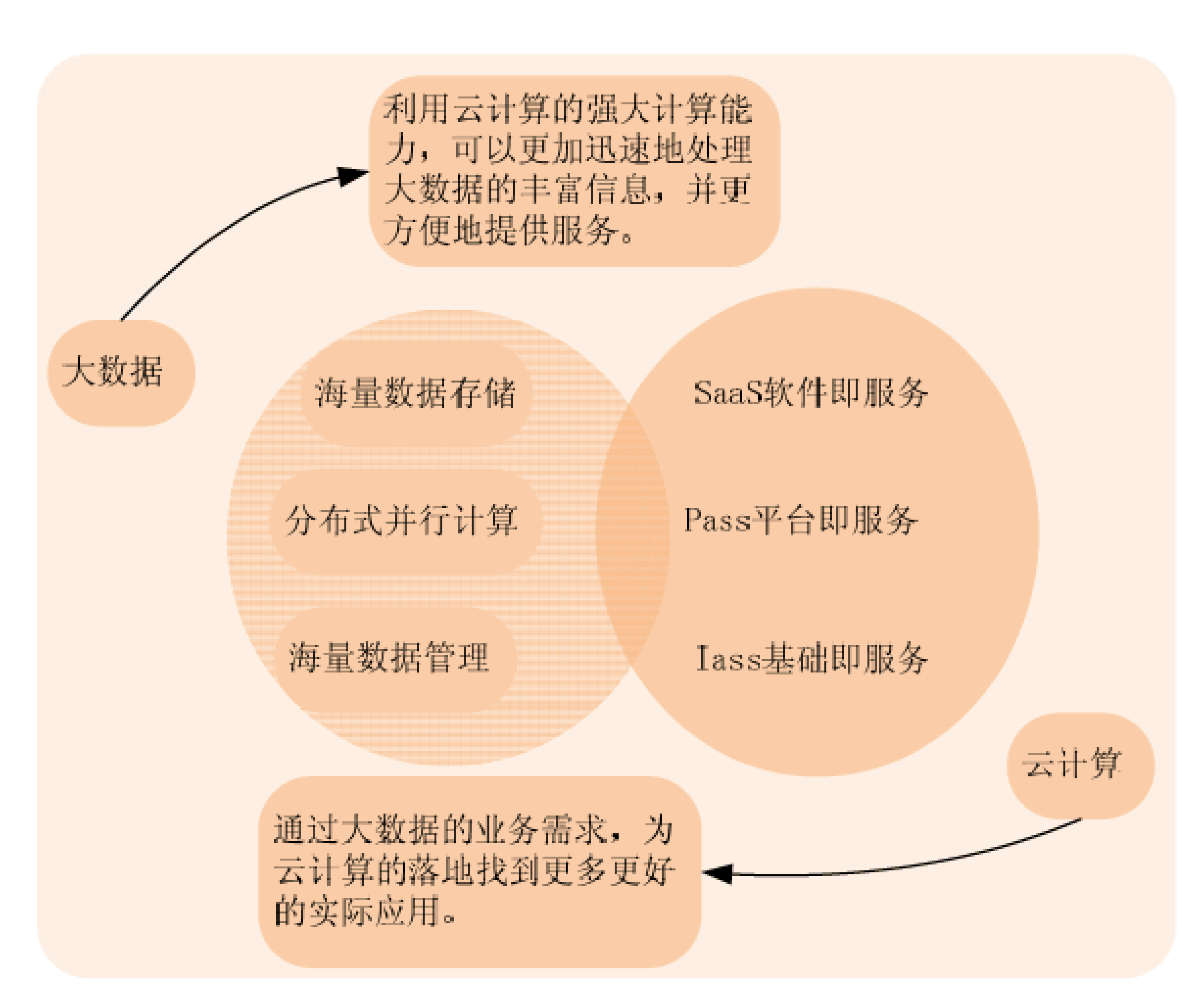


图 3-3　大数据与云计算关系

从二者的定义范围来看，大数据要比云计算更加广泛。大数据这一概念从 2011 年诞生以来，历经 5 个年头。大数据是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力来适应海量、高增长率和多样化的信息资产。大数据这个强大的数据库拥有三层架构体系，包括数据存储、处理与分析。简而言之，数据需要通过存储层先存储下来，之后根据要求建立数据模型体系，进行分析产生相应价值。这其中缺少不了云计算所提供的中间数据处理层强大的并行计算和分布式计算能力。

据了解，云计算的历史比大数据更加绵长，是继 1980 年大型计算机到客户端服务器转变之后的一种巨变。美国国家标准与技术研究院定义云计算为一种按使用量付费的模式，这种模式提供可用的、便捷的、按需的网络访问。同时，进入可配置的计算资源共享池即可快速提供资源，减少交互所需的步骤和时间。云计算可以实现每秒 10 万亿次的运算，能够模拟核爆炸，分析市场发展趋势，预测气候变化等。所以，云计算的作用和大数据类似，云计算与大数据如同手心手背的关系：二者不可或缺，相辅相成。没有大数据，云计算没有用武之地，而没有云计算成就不了大数据。

以此看来，大数据与云计算之间，并非独立概念，而是关系非比寻常。无论在资源的需求上还是在资源的再处理上，都需要二者共同运用。因此，与其计较大数据与云计算之间怎么区分，不如规划在一起，让云计算为大数据提供强大平台，以大数据分析出的结论完成云计算价值。

3.5 数据向云计算迁移

尽管迁移到云的数据中心设施有着诸多的原因和好处，但这一过程仍然是充满了各种风险。以前，当需要结束一段与数据中心或云服务提供商的失败的合作关系或者服务提供者本身出现故障的时候，我们需要一套“B 计划”。通常的建议是，IT 管理人员应确保他们所在的企业和云服务提供商之间的合同内要规定好数据的所有权是属于企业的。然而，这只是问题的一部分，依然存在如何在事后处理数据的问题。

随着数据正在逐渐成为企业的核心资产，数据的存储、迁移成为企业时刻关注的问题。据 Gartner 统计数据表明，95%的受访企业都认为数据迁移是个“硬骨头”，让人头痛却又不得不面对。

在传统数据迁移的过程中，有接近 64%的客户经历过业务下线时间超过计划和预期的情况；超过 50%的客户在实施数据迁移过程中，将面临技术升级或是架构革新所带来的兼容性风险；而在架构升级之后，性能不升反降、数据受损、数据丢失等，如梦魇一般，也时刻困扰着实施数据迁移行为的客户。

数据迁移是一个非常复杂和细致的过程，它要求设计与实施人员具备多元化的知识结构，否则，我们就不得不陷入顾此失彼的境地。一个完整和成功的数据迁移解决方案包含 6 个部分：

- 具备专业技能的人员。

- 经过验证的解决方案。
- 稳健可靠的流程和方法论。
- 交付前实验室模拟验证。
- 高效的迁移工具。
- 应急的回滚措施。

任意一个条件的缺乏都有可能导致数据迁移不成功。以华为融合数据迁移为例，融合数据迁移解决方案自上而下，支持基于应用软件与数据库、卷管理软件、虚拟机、主机、网络，以及存储 6 种层级功能的数据迁移方案，从不同维度、不同层面，保障我们客户的数据迁移过程可靠与高效。图 3-4 为数据迁移建模设计。

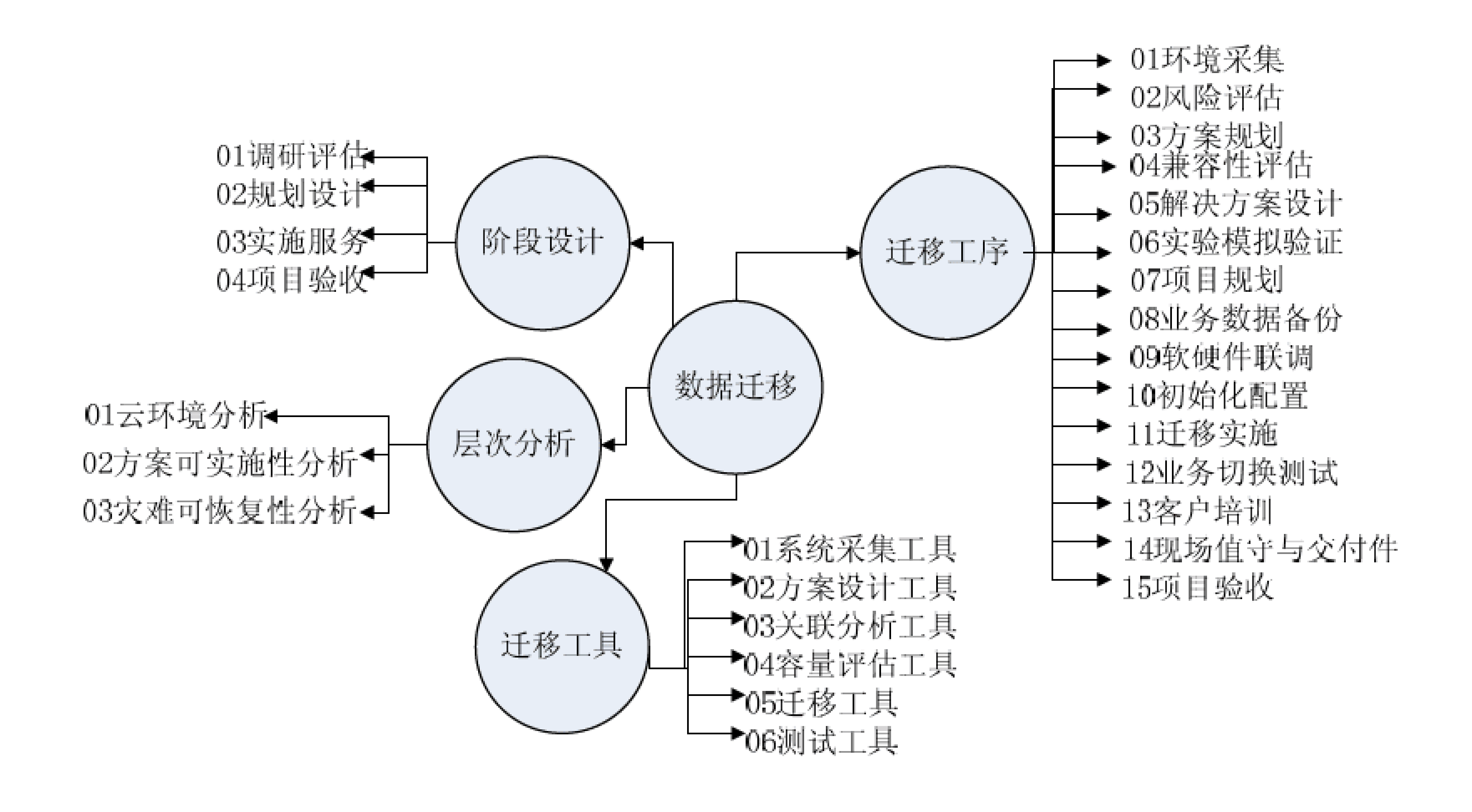


图 3-4　数据迁移建模设计

融合数据迁移解决方案分为四大阶段与 15 道工序，从调研评估到规划设计，再到实施服务，以至于到最后的项目验收，将严格地遵循这样的服务流程与方法论，为客户提供规范的、标准的、无差别的高质量数据迁移。

在方案设计阶段提供深入、细致的三次业务分析用于建模。对现网环境及云环境分析，将有助于工程师全面、细致地掌握客户现场情况，以应对后续的兼容性等多方面风险；方案可实施性分析，为用户后续迁移工作的有序开展提供必要的数据支撑；风险分析将作为最关键的一道保护措施，在数据迁移出现不可接受的偏差时，及时、安全、有序地恢复用户业务。数据迁移完成后检验数据移动的情况，通过特有的校验算法来验证数据的一致性，保证数据的完整性、正确性、有效性，增强客户对关键信息安全性的分析。

3.6 大数据清洗

数据清洗是指发现并纠正数据文件中可识别的错误的最后一道程序，包括检查数据一致性、处理无效值和缺失值等。与问卷审核不同，录入后的数据清理一般是由计算机而不是人

工完成。数据清洗需要注意的是不要将有用的数据过滤掉，对于每个过滤规则认真进行验证，并要用户确认。数据清洗从名字上也看得出就是把“脏”的“洗掉”，指发现并纠正数据文件中可识别的错误的最后一道程序，包括检查数据一致性、处理无效值和缺失值等[35]。

1. 一致性检查

一致性检查（consistency check）是根据每个变量的合理取值范围和相互关系，检查数据是否合乎要求，发现超出正常范围、逻辑上不合理或者相互矛盾的数据。SPSS、SAS 和 Excel 等计算机软件都能够根据定义的取值范围，自动识别每个超出范围的变量值。具有逻辑上不一致性的答案可能以多种形式出现：例如，许多调查对象说自己开车上班，又报告没有汽车；或者调查对象报告自己是某品牌的重度购买者和使用者，但同时又在熟悉程度量表上给了很低的分值。发现不一致时，要列出问卷序号、记录序号、变量名称、错误类别等，便于进一步核对和纠正。

2. 无效值和缺失值的处理

由于调查、编码和录入误差，数据中可能存在一些无效值和缺失值，需要给予适当的处理。常用的处理方法有：估算、整例删除、变量删除和成对删除。

估算（estimation）。最简单的办法就是用某个变量的样本均值、中位数或众数代替无效值和缺失值。这种办法简单，但没有充分考虑数据中已有的信息，误差可能较大。另一种办法就是根据调查对象对其他问题的答案，通过变量之间的相关分析或逻辑推论进行估计。例如，某一产品的拥有情况可能与家庭收入有关，可以根据调查对象的家庭收入推算拥有这一产品的可能性。

整例删除（casewise deletion）是剔除含有缺失值的样本。由于很多问卷都可能存在缺失值，这种做法的结果可能导致有效样本量大大减少，无法充分利用已经收集到的数据。因此，只适合关键变量缺失，或者含有无效值或缺失值的样本比重很小的情况。

变量删除（variable deletion）。如果某一变量的无效值和缺失值很多，而且该变量对于所研究的问题不是特别重要，则可以考虑将该变量删除。这种做法减少了供分析用的变量数目，但没有改变样本量。

成对删除（pairwise deletion）是用一个特殊码（通常是 9、99、999 等）代表无效值和缺失值，同时保留数据集中的全部变量和样本。但是，在具体计算时只采用有完整答案的样本，不同的分析因涉及的变量不同，其有效样本量也会有所不同。这是一种保守的处理方法，最大限度地保留了数据集中的可用信息。

采用不同的处理方法可能对分析结果产生影响，尤其是当缺失值的出现并非随机且变量之间明显相关时。因此，在调查中应当尽量避免出现无效值和缺失值，保证数据的完整性。

无论用海量数据还是大数据来表征这个时代，数据规模庞大、增长迅速、类型繁多、结构各异已成为无法回避的现实问题。如何把繁杂的大数据变成我们能应付的、有效的“小”数据，即针对特定问题而构建一个干净、完备的数据集，这一过程变得尤为重要。在大数据时代，若不加强数据清洗，则 GIGO（垃圾进，垃圾出）现象会更加严重。对数据的清洗之后进行分析挖掘的过程就是情报“去粗取精、去伪存真、化零为整、见微知著”的过程。只有通过清洗与过滤得到干净完备的数据，才能通过分析与挖掘得到可以让人放心的、可用于

支撑决策的情报。有时决策者似乎只需要一个简单的数，但是为了得到这一个数，我们需要搜集大量数据并进行有效的分析与处理。

3.7 云计算时代的数据集成技术

随着公有云平台和私有云平台的流行，数据集成问题越来越重要。以往存储在企业内部的信息，现在要分散在不同的公有云平台上。这些信息，要进行广泛的共享。企业内部部署的系统、公有云平台上部署的系统，彼此之间都需要共享信息。

到目前为止，数据集成焦点还集中于现有的集成技术，包括传统的和非传统的。 所执行的任务包括：数据复制、语义解析、数据清洗、海量数据迁移。这些技术帮助企业在云—云之间、云—企业之间，或者企业—企业之间传输数据，以支持核心业务流程。 过去的几年里，这些技术一直在演进以适应混合云以及多云架构，当然还要适应大数据集的出现。随着云计算技术的成熟，数据集成将呈现新的形式、扮演新的角色，并贡献新的价值。数据在云上存储，和在非云系统上存储的方法不同。新的数据集成的功能应能够分别处理这两种存储方法，并高效地进行数据结构和内容的处理，从而让目标系统如同访问本地数据一样。海量数据迁移包括 ETL（抽取－转换－加载）功能，并包括海量数据的定时迁移、内容和结构的变更，以满足目标系统，例如云数据仓库的需要。数据清洗技术，使数据集成过程中能够去掉或者改正错误的和不准确的数据。 在数据集成的其他操作中，尤其是从一个系统传输信息到另一系统时，都需要进行数据清洗。随着云计算成为企业的主流平台，数据集成的世界也需要跟进，要开拓和扩展新的能力。

（1）智能数据服务搜索，是一种数据集成技术，能够自动发现和定义数据服务。这种技术将成为云计算和非云计算系统生产数据和消费数据的主流机制。就是说，可以搜寻到或者重新搜寻到企业内部存在的数据服务。更重要的是，搜寻到公有云上的数据服务，找到数据服务的位置、提供的功能，以及如何访问这些数据服务。企业就能够利用这些编目来理解所有可用的数据资产，并利用这些有用的数据资产来支持核心业务流程。

（2）数据虚拟化，企业希望利用新的虚拟化结构来重新定义现有的数据库，并把这些数据库以定义好的数据服务的形式，提供给外部。完全可以用新的虚拟数据库结构，置于现存数据库之上，从而重新定义数据库的访问方式。这样一来，就不需要冒险去重新构建后台数据库，便可以满足云计算系统的需要。

（3）数据编排，和服务编排类似，定义数据之间进行交互以形成方案的能力。定义混合的数据点，也许是销售和客户，以形成新的数据服务来服务企业内部和外部用户。这样，使用数据的用户，将能够更好控制数据对每一应用视图的用途，而不必更改数据的物理结构和内容。

（4）数据标识，从结构和实例两个角度连接数据到用户和机器的能力。主要控制谁和什么系统能够消费数据，并看到内容。这对适应各种变更和扩展的法规，以及各种内部数据安全策略来说，带来很大便利。数据容器控制对数据的访问，以及设置在数据中的数据标识规则。这将会是一个通用的机制，用在企业和公有云提供者之间。

（5）身份识别和集中信用也是控制数据访问权限机制的新一代技术。通过提供集中化的位置去验证数据（结构和内容），验证要访问和操作数据的用户和设备，把数据标识提高到另一个层次。这种机制意味着能够了解数据存放的位置，并将授权用户匹配到授权的数据，从数据库、对象到实例。

3.8 云推荐

云推荐是一款面向网站的高速、稳定、易用、免费的站内个性智能推荐系统。云推荐[36]由国内著名统计技术服务商 CNZZ 于 2012 年 11 月推出一款智能站内内容推荐系统。云推荐运用了云计算（Cloud Computing）技术的推荐引擎。该推荐引擎和一般的推荐引擎不同，云推荐引擎不需要部署在网站的服务器上，而是统一部署在第三方的云计算平台上。

目前国内阿里巴巴公司和 CNZZ 合作，运用阿里云先进的云计算系统，支持海量网页数据和个性用户行为分析；同时依托 CNZZ 网站统计提供多角度的专业数据分析报表，达到帮助网站向用户精准推送个性内容、提升网站流量、加强用户黏性的目的。

因此，网站不需要付出任何硬件、带宽的成本，也免除了推荐系统的开发和维护代价，只需要部署云推荐引擎的一段 js 代码，就可以实现站内文章和热词智能推荐，极大地降低了网站的使用和开发成本。

云推荐通过部署在网站上的 js 代码动态展示针对当前页面的推荐效果，纯异步加载，不会影响网页打开速度，同时收集用户的访问历史，自动计算文章的热门度、文章之间的关联度、用户的阅读历史、兴趣推测、用户兴趣类聚、文章主题词提取等，基于这些算法的不同组合，针对不同网站提供不同的最佳推荐效果。

由于各个网站共享同一个推荐引擎平台，所以云推荐必须基于云计算技术来处理海量数据计算，不仅互联网上的不同网站的文章集合是海量的，而且用户集合也是海量的，这样的计算规模已经不能通过单机或者简单的多机计算来处理，必须依赖成熟的分布式存储（DFS）和并行计算框架（Map-Reduce）来支持。同样，推荐技术离不开搜索技术的支持，在关键词匹配算法中，采取和搜索引擎类似的排序逻辑，整体系统和搜索有相似之处。

云推荐的技术优势与特点说明如下。

- 结合 CNZZ 统计分析，贴合用户轨迹推荐最佳内容。
- 用户个性行为与全网分析结合，有效盘活网站长尾、新老数据。
- 多样性算法支持，涵盖关键词、用户行为、数据分类等领域。
- 根据网站特点，智能调整各种贡献内容比例，实现个性化推荐。
- 智能推荐网页内容，引发新页面曝光、快速提升网站流量。
- 了解用户最感兴趣的内容，提高用户访问深度，增加用户黏性。
- 提供专业的深度分析报表，实时监控推荐数据。
- 提供丰富的推荐样式模版，支持自定义设置，保持风格统一。

3.9 本章小结

云计算正向大数据延伸，如若说云计算铺好了路桥，大数据就是在这些信息高速路上行走的车辆。云计算向大数据延伸的特征就是数据的智能化应用。数据的快速增长带来了数据存储、处置、剖析的巨大压力，以数据智能剖析为特征的大数据技术引入，成为企业提升竞争力的有力工具。大数据技术与云计算的发展亲密相关，大数据技术是云计算技术的延伸。大数据技术涵盖了从数据的海量存储、处置到应用多方面的技术，包括海量分布式文件系统、并行计算框架、非 SQL 数据库、实时流数据处置以及智能剖析技术（如模式识别、自然语言理解、应用知识库）等。随着社会化网络、移动支付以及物联网的发展，实体经济和虚拟世界有更多的交集，数据的价值将不停地提升。

本章介绍了大数据与云的融合关系、云和大数据的安全隐私问题、大数据基于云的商业价值、大数据向云的迁移、大数据清洗及精准推荐等基于云的大数据延展技术。

第 4 章

Spark大数据处理基础

Spark 是一个基于内存计算的开源集群计算系统，目的是更快速地进行数据分析。Spark 由加州伯克利大学 AMP 实验室 Matei 为主的团队使用 Scala 开发，其核心部分的代码只有 63 个 Scala 文件，非常轻量级[37]。Spark 提供了与 Hadoop 相似的开源集群计算环境，但基于内存和迭代优化的设计，Spark 在某些工作负载表现更优秀。到目前为止，Spark 开源生态系统得到了大幅增长，已成为大数据领域最活跃的开源项目之一，当下已活跃在 Hortonworks、IBM、Cloudera、MapR 和 Pivotal 等众多知名大数据公司。本章主要介绍 Spark 大数据计算框架、架构、计算模型和数据管理策略及 Spark 在工业界的应用。

4.1 Spark 大数据处理技术

Spark 拥有 Hadoop Map-Reduce 所具有的优点，但不同于 Map-Reduce 的是 Job 中间输出和结果可以保存在内存中，从而不再需要读写 HDFS，因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 Map-Reduce 的算法。Spark 基于内存和迭代优化的设计，使 Spark 在交互式数据分析和数据挖掘工作负载中表现得更加优秀。

4.1.1 Spark 系统概述

Spark 基于内存计算，提高了在大数据环境下数据处理的实时性，同时保证了高容错性和高可伸缩性，允许用户将 Spark 部署在大量廉价硬件之上，形成集群。Spark 于 2009 年诞生于加州大学伯克利分校 AMPLab。目前，已经成为 Apache 软件基金会旗下的顶级开源项目[38]。

AMPLab 开发以 Spark 为核心的 BDAS 时提出的目标是：one stack to rule them all，也就是说在一套软件栈内完成各种大数据分析任务。相对于 MapReduce 上的批量计算、迭代型计算以及基于 Hive 的 SQL 查询，Spark 可以带来上百倍的性能提升。目前 Spark 的生态系统日趋完善，Spark SQL 的发布、Hive on Spark 项目的启动以及大量大数据公司对 Spark 全栈的支持，让 Spark 的数据分析范式更加丰富。

Spark 是一个计算框架，而 Hadoop 中包含计算框架 MapReduce 和分布式文件系统 HDFS，Hadoop 更广泛地说还包括在其生态系统上的其他系统，如 HBase、Hive 等。Spark 是 MapReduce 的替代方案，而且兼容 HDFS、Hive 等分布式存储层，可融入 Hadoop 的生态系统，以弥补缺失 MapReduce 的不足。Spark 的一站式解决方案有很多的优势[39]：

1. 全栈多计算范式的高效数据流水线

Spark 支持复杂查询。Spark 还支持 SQL 查询、流式计算、机器学习和图算法。同时，用户可以在同一个工作流中无缝搭配这些计算范式。

2. 轻量级快速处理

Scala 语言的简洁和丰富的表达力，以及 Spark 充分利用和集成 Hadoop 等其他第三方组件，同时着眼于大数据处理，Spark 通过将中间结果缓存在内存减少磁盘 I/O 来达到性能的提升。

3. Spark 支持多语言

Spark 支持通过 Scala、Java 及 Python 编写程序，这允许开发者在自己熟悉的语言环境下进行工作。它自带了 80 多个算子，同时允许在 Shell 中进行交互式计算。用户可以利用 Spark 像书写单机程序一样书写分布式程序，轻松利用 Spark 搭建大数据内存计算平台并充分利用内存计算，实现海量数据的实时处理。

4. 与 HDFS 等存储层兼容

Spark 可以运行在任何 Hadoop 数据源上，如 Hive、HBase、HDFS 等。这个特性让用户可以轻易迁移已有的持久化层数据。

5. 任务调度的开销小

Spark 采用了事件驱动的类库 AKKA 来启动任务，通过线程池复用线程来避免进程或线程的启动和切换开销。

6. 优化的数据存储

Spark 抽象出分布式内存存储结构弹性分布式数据集（RDD），进行数据的存储。RDD 支持粗粒度写操作，但对于读取操作，RDD 可以精确到每条记录，这使得 RDD 可以用来作为分布式索引。

4.1.2 Spark 生态系统 BDAS（伯利克分析栈）

目前，Spark 已经发展成为包含众多子项目的大数据计算平台。伯克利将 Spark 的整个生态系统称为伯克利数据分析栈（BDAS）。其核心框架是 Spark，同时 BDAS 涵盖支持结构化数据 SQL 查询与分析的查询引擎 Spark SQL 和 Shark，提供机器学习功能的系统 MLbase 及底层的分布式机器学习库 MLlib、并行图计算框架 GraphX、流计算框架 Spark Streaming、采样近似计算查询引擎 BlinkDB、内存分布式文件系统 Tachyon、资源管理框架 Mesos 等子项目。这些子项目在 Spark 上层提供了更高层、更丰富的计算范式。图 4-1 所示为 Spark 生态系统项目结构图。

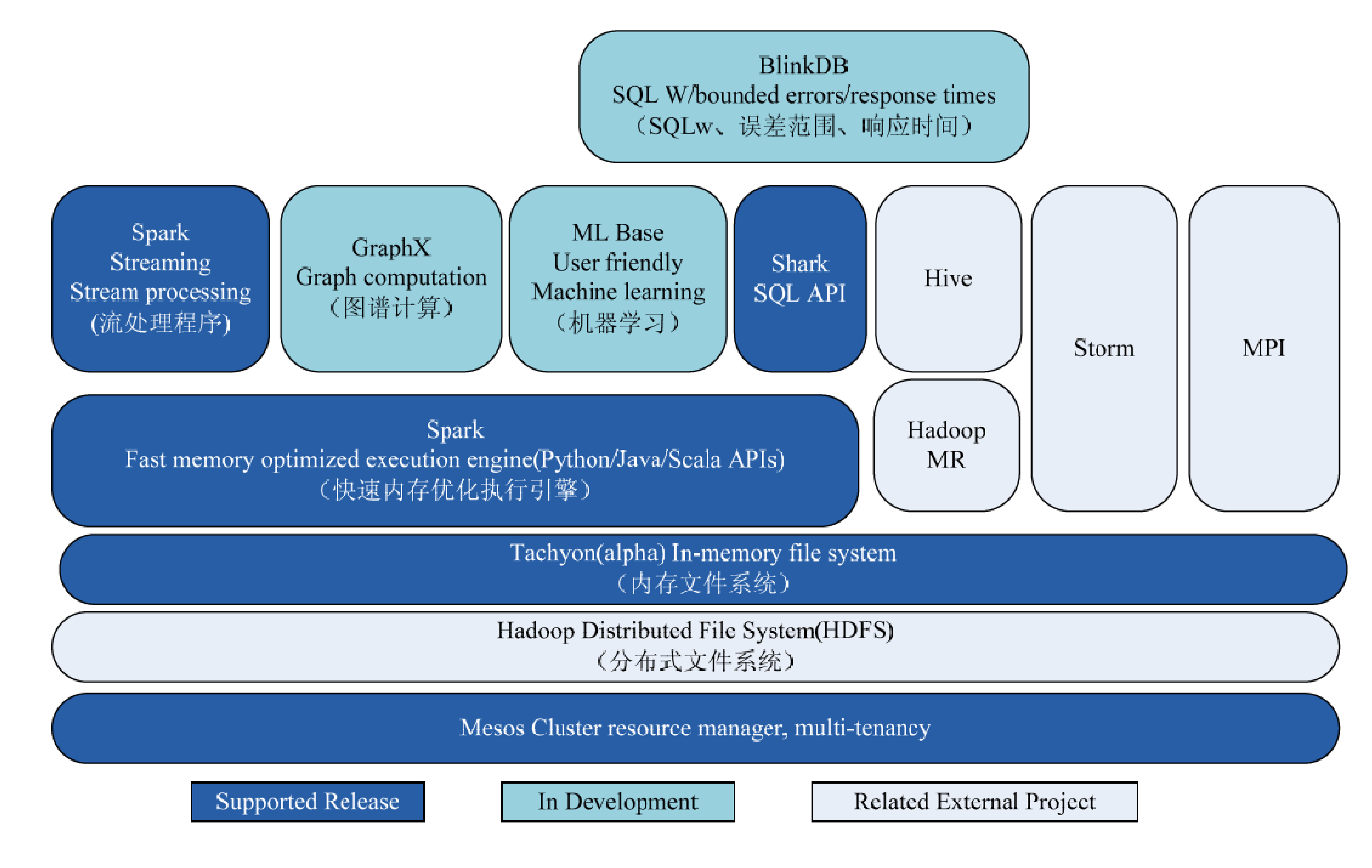


图 4-1　Spark 生态系统项目结构图

4.1.3　Spark 的用武之地

Spark 引进了弹性分布式数据集 RDD （Resilient Distributed Dataset)）的抽象，它是分布在一组节点中的只读对象集合，这些集合是弹性的，如果数据集一部分丢失，则可以根据“血统”（即允许基于数据衍生过程重建部分数据集的信息）对它们进行重建[39]。另外在 RDD 计算时可以通过 CheckPoint 来实现容错，而 CheckPoint 有两种方式：CheckPoint Data 和 Logging The Updates，用户可以控制采用哪种方式来实现容错[40]。

Spark 提供的数据集操作类型有很多种，大致分为：Transformations 和 Actions 两大类。Transformations 包括 Map、Filter、FlatMap、Sample、GroupByKey、ReduceByKey、Union、Join、Cogroup、MapValues、Sort 和 PartionBy 等多种操作类型，同时还提供 Count；Actions 包括 Collect、Reduce、Lookup 和 Save 等操作。另外各个处理节点之间的通信模型不再像 Hadoop 只有 Shuffle 一种模式，用户可以命名、物化，控制中间结果的存储、分区等[41]。

目前大数据处理场景有以下几个类型：

（1）复杂的批量处理（Batch Data Processing），偏重点在于处理海量数据的能力，至于处理速度可忍受，通常的时间可能是在数十分钟到数小时。

（2）基于历史数据的交互式查询（Interactive Query），通常的时间在数十秒到数十分钟之间。

（3）基于实时数据流的数据处理（Streaming Data Processing），通常在数百毫秒到数秒之间。

目前对以上三种场景需求都有比较成熟的处理框架，第一种情况可以用 Hadoop 的 MapReduce 来进行批量处理海量数据，第二种情况可以用 Impala 进行交互式查询，对于第三种情况可以用 Storm 分布式处理框架处理实时流式数据。以上三者都相对独立、各自一套，维护成本比较高，而 Spark 的出现能够一站式平台满足以上需求。

通过以上分析，总结出 Spark 使用场景有以下几个[42,43]：

（1）Spark 是基于内存的迭代计算框架，适用于需要多次操作特定数据集的应用场合。需要反复操作的次数越多，所需读取的数据量越大，受益越大。数据量小但是计算密集度较大的场合，受益就相对较小。

（2）由于 RDD 的特性，Spark 不适用那种异步细粒度更新状态的应用，例如 Web 服务的存储或者是增量的 Web 爬虫和索引，即对于那种增量修改的应用模型不适合。

（3）数据量不是特别大，但是要求实时统计分析需求。

4.1.4 Spark 大数据处理框架

Spark 是整个 BDAS 的核心，生态系统中的各个组件通过 Spark 来实现对分布式并行任务处理的程序支持[44]。Spark 架构采用了分布式计算中的 Master-Slave 模型。Master 是对应集群中含有 Master 进程的节点，Slave 是集群中含有 Worker 进程的节点。Spark 结构图如图 4-2 所示。

- Master 作为整个集群的控制器，负责整个集群的正常运行。
- Worker 相当于是计算节点，接收主节点命令与进行状态汇报。
- Executor 负责任务的执行。
- Client 作为用户的客户端负责提交应用。
- Driver 负责控制一个应用的执行。

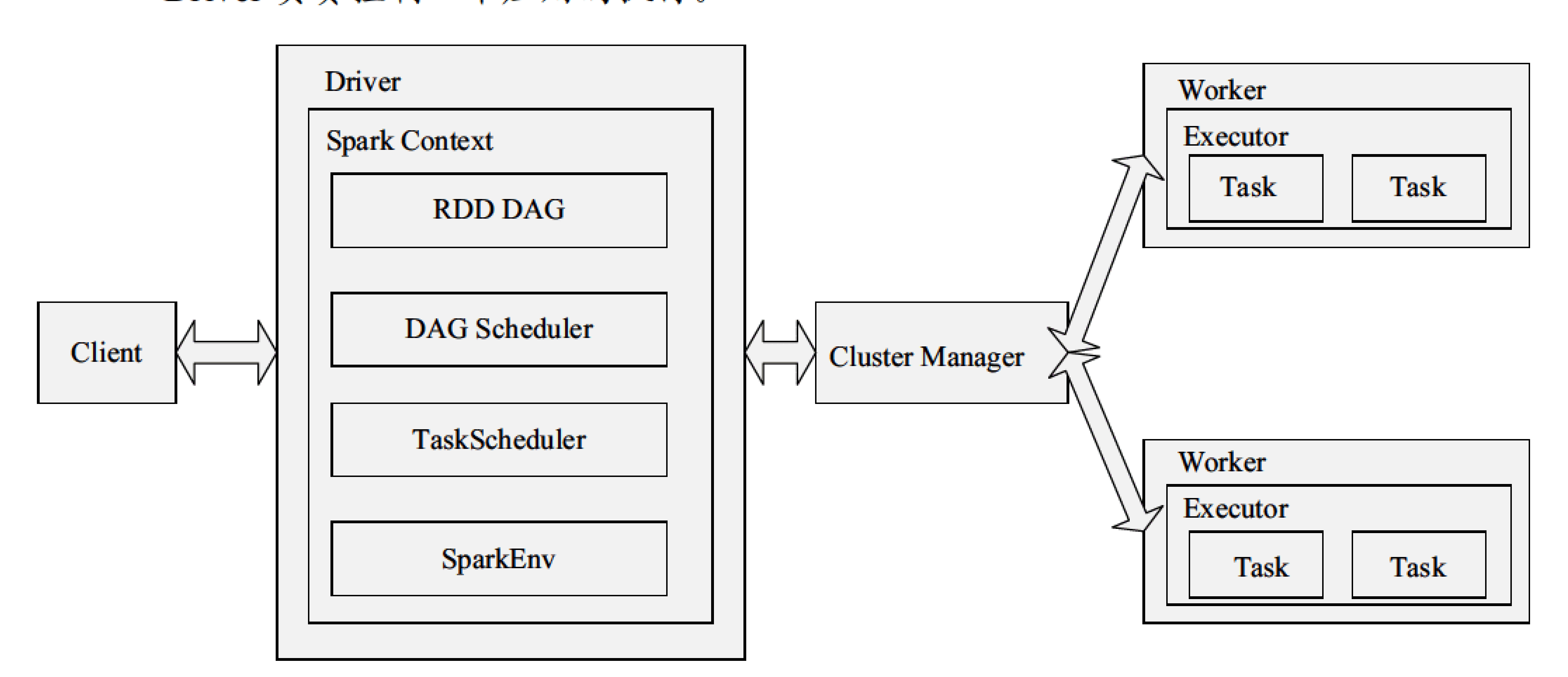


图 4-2 Spark 结构图

4.1.5　Spark 运行模式分类及术语

Spark 的运行模式多种多样、灵活多变，部署在单机上时，既可以用本地模式运行，也可以用伪分布式模式运行。而当以分布式集群的方式部署时，也有众多的运行模式可供选择，这取决于集群的实际情况。底层的资源调度既可以依赖于外部的资源调度框架，也可以使用 Spark 内建的 Standalone 模式。对于外部资源调度框架的支持，目前的实现包括相对稳定的 Mesos 模式，以及还在持续开发更新中的 Hadoop YARN 模式[45,46]。

在实际应用中，Spark 应用程序的运行模式取决于传递给 SparkContext 的 Master 环境变量的值，个别模式还需要依赖辅助的程序接口来配合使用，目前所支持的 Master 环境变量由特定的字符串或 URL 所组成。表 4-1 给出 Spark 运行模式，表 4-2 给出 Spark 常用术语。

表 4-1　Spark 运行模式

运行环境	模式	描述
Local	本地模式	常用于本地开发测试，本地还分为 local 单线程和 local-cluster 多线程
Standalone	集群模式	典型的 Mater/slave 模式，不过也能看出 Master 是有单点故障的，Spark 支持 ZooKeeper 来实现 HA
On yarn	集群模式	运行在 YARN 资源管理器框架之上，由 YARN 负责资源管理，Spark 负责任务调度和计算
On mesos	集群模式	运行在 Mesos 资源管理器框架之上，由 Mesos 负责资源管理，Spark 负责任务调度和计算
On cloud	集群模式	比如 AWS 的 EC2，使用这个模式能很方便地访问 Amazon 的 S3；Spark 支持多种分布式存储系统：HDFS 和 S3

表 4-2　Spark 常用术语

术语	描述
Application	Spark 的应用程序，包含一个 Driver program 和若干 Executor
SparkContext	Spark 应用程序的入口，负责调度各个运算资源，协调各个 Worker Node 上的 Executor
Driver Program	运行 Application 的 main()函数并且创建 SparkContext
Executor	是为 Application 运行在 Worker node 上的一个进程，该进程负责运行 Task，并且负责将数据存在内存或者磁盘上。 每个 Application 都会申请各自的 Executor 来处理任务
Cluster Manager	在集群上获取资源的外部服务 (例如：Standalone、Mesos、YARN)
Worker Node	集群中任何可以运行 Application 代码的节点，运行一个或多个 Executor 进程
Task	运行在 Executor 上的工作单元
Job	SparkContext 提交的具体 Action 操作，常和 Action 对应
Stage	每个 Job 会被拆分成很多组 task，每组任务被称为 Stage，也称 TaskSet
RDD	RDD 是 Resilient Distributed Datasets 的简称，中文为弹性分布式数据集，它是 Spark 最核心的模块和类
DAGScheduler	根据 Job 构建基于 Stage 的 DAG，并提交 Stage 给 TaskScheduler
TaskScheduler	将 Taskset 提交给 Worker node 集群运行并返回结果
Transformations	是 Spark API 的一种类型，Transformation 返回值还是一个 RDD。 所有的 Transformation 采用的都是懒策略，如果只是将 Transformation 提交是不会执行计算的
Action	是 Spark API 的一种类型，Action 返回值不是一个 RDD，而是一个 Scala 集合。计算只有在 Action 被提交的时候计算才被触发

其实这些运行模式尽管表面上看起来差异很大，但总体上来说，都基于一个相似的工作流程。它们从根本上都是将 Spark 的应用分为任务调度和任务执行两个部分，上面图 4-2 所示的是在分布式模式下，Spark 的各个调度和执行模块的大致框架。对于本地模式来说，其内部程序逻辑结构也是类似的，只是其中部分模块有所简化。

4.2 Spark 2.0.0 安装配置

Spark 最早是为了在 Linux 平台上使用而开发的，在生产环境中也是部署在 Linux 平台上，但是 Spark 在 UNIX、Windwos 和 Mac OS X 系统上也运行良好。不过，在 Windows 上运行 Spark 稍显复杂，必须先安装 Cygwin 以模拟 Linux 环境，才能安装 Spark。

Spark 的安装简便，用户可以在官网上下载到最新的软件包，网址为 http：//spark.apache.org/。由于 Spark 主要使用 HDFS 充当持久化层，所以完整地使用 Spark 需要预先安装 Hadoop。

Spark 在生产环境中，主要部署在安装有 Linux 系统的集群中。在 Linux 系统中安装 Spark 需要预先安装 JDK、Scala 等所需的依赖。由于 Spark 是计算框架，所以需要预先在集群内有搭建好存储数据的持久化层，如 HDFS、Hive、Cassandra 等。最后用户就可以通过启动脚本运行应用了。下面介绍 Spark 2.0.0 版本集群的安装和部署。

4.2.1 在 Linux 集群上安装与配置 Spark

从 Spark 项目网站 http://spark.apache.org 下载页面获取 Spark。本章采用 Spark 2.0.0 版本。Spark 采用 Hadoop 的客户端库的 HDFS 和 YARN，下载预打包一些流行的 Hadoop 版本。用户还可以下载“免费 Hadoop”，并通过扩充 Spark 的类路径，使用任何 Hadoop 版本运行 Spark。Spark 下载页面如图 4-3 所示。

Spark 可运行在 Windows 和 UNIX 系统（如 Linux、Mac OS）上，很容易在本地已经安装 Java 的计算机上运行 Spark 系统，所以你需要的是有 Java 安装在你的系统路径，或 JAVA_HOME 环境变量指向一个 Java 安装。

Spark 运行在 Java 7+、Python 2.6+/3.4+和 3.1+。对于 Scala API，Spark 2.0.0 使用 Scala 2.11，你将需要使用一个兼容的 Scala 版本（2.11.X）。Spark 2.0.0 默认 Scala 版本为 2.11。Scala 2.10 用户要下载 Spark 源码包并以 Scala 2.10 构建 Spark 支撑。

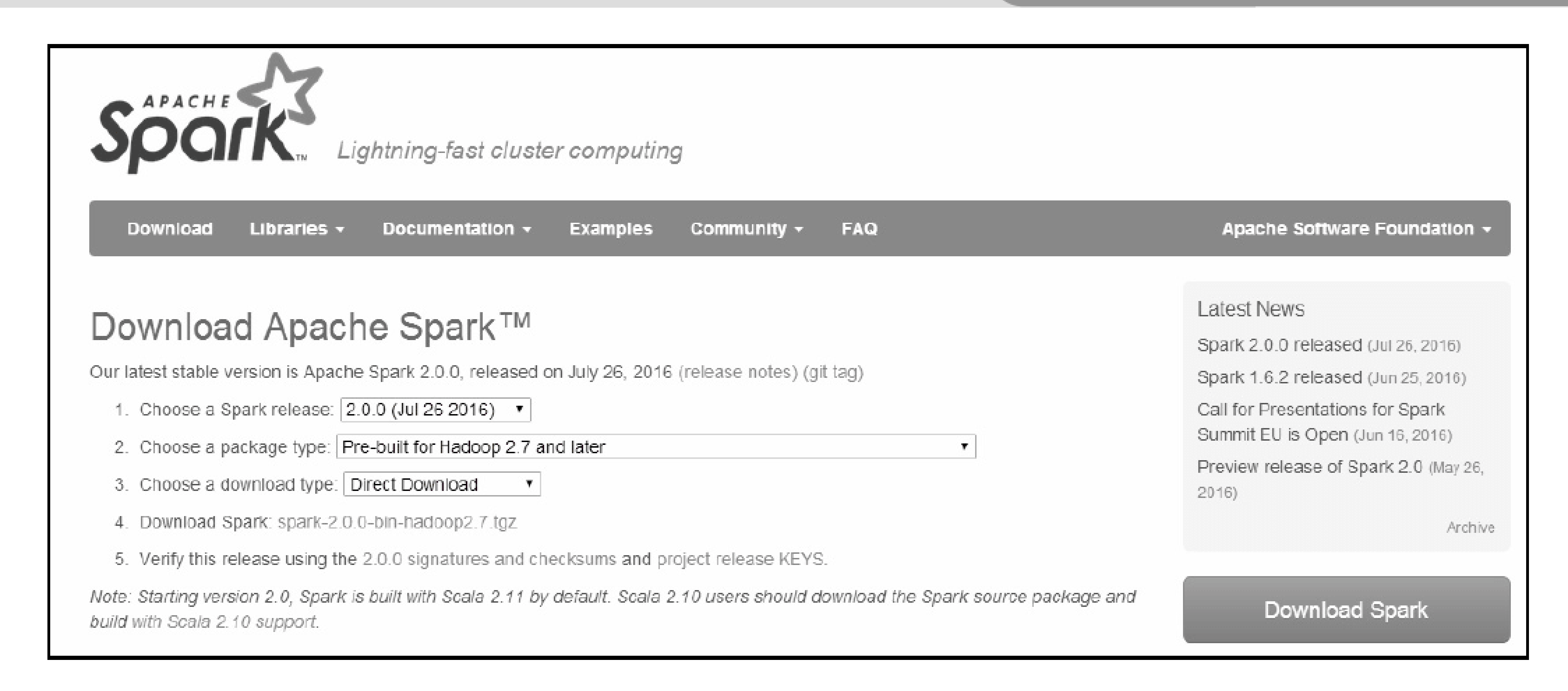


图 4-3　Spark 下载页面

实验使用的环境是安装 Ubuntu 14.04 LTS 两台计算机，一个当作 Master，一个当作 Slave。

- JDK 版本：jdk1.7.0_60。
- Scala 版本：Scala-2.11。
- Hadoop 版本：Hadoop 2.7。
- Spark 版本：spark-2.0.0。

1．安装 JDK

安装 JDK 大致分为下面 4 个步骤。

（1）用户可以在 Oracle JDK 的官网下载相应版本的 JDK，本例以 jdk1.7.0_60 为例，官网地址为 http：//www.oracle.com/technetwork/java/javase/downloads/index.html。

（2）下载后，在解压出 JDK 的目录下执行 bin 文件。

（3）配置环境变量，在/etc/profile 增加以下代码。

```
JAVA_HOME=/home/gcs/user/java/jdk1.7.0_60
PATH= $JAVA_HOME/bin:$PATH
CLASSPATH=.: $JAVA_HOME/jre/lib/rt.jar: $JAVA_HOME/jre/lib/dt.jar: $JAVA_HOME/ jre/lib/tools.jar
export JAVA_HOME PATH CLASSPATH
```

（4）使 profile 文件更新生效。

```
./etc/profile
```

以下为 JDK 安装代码：

```
yum search openjdk-devel
sudo yum install java-1.7.0_60-openjdk-devel.x86_64
```

```
/usr/sbin/alternatives --config java
/usr/sbin/alternatives --config javac
sudo vim /etc/profile
# add the following lines at the end
export JAVA_HOME=/usr/lib/jvm/java-1.7.0_60-openjdk-1.7.0.60.x86_64
export JRE_HOME=$JAVA_HOME/jre
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
# save and exit vim
# make the bash profile take effect immediately
$ source /etc/profile
# test
$ java -version
```

2. 安装 Scala

Scala 官网提供各个版本的 Scala，用户需要根据 Spark 官方规定的 Scala 版本进行下载和安装。Scala 官网地址为 http：//www.scala-lang.org/。本章以 Scala-2.11 为例。

（1）下载 scala-2.11.1.tgz。

（2）在目录下解压。

```
tar -xzvf scala-2.11.1.tgz。
```

（3）配置环境变量，在/etc/profile 中添加下面的内容。

```
export SCALA_HOME=/home/chengxu/scala-2.11.1/scala-2.11.1
export PATH=${SCALA_HOME}/bin: $PATH
```

（4）使 profile 文件更新生效。

```
./etc/profile
```

以下为 Scala 安装代码：

```
$ tar -zxf scala-2.11.1.tgz
$ sudo mv scala-2.11.1 /usr/lib
$ sudo vim /etc/profile
# add the following lines at the end
export SCALA_HOME=/usr/lib/scala-2.11.1
export PATH=$PATH:$SCALA_HOME/bin
# save and exit vim
#make the bash profile take effect immediately
source /etc/profile
# test
```

```
11. $ scala -version
```

3. 配置 SSH 免密码登录

Spark 的 Master 节点向 Worker 节点发命令需要通过 ssh 进行发送，用户不希望 Master 每发送一次命令就输入一次密码，因此需要实现 Master 无密码登录到所有 Worker。Master 作为客户端，要实现无密码公钥认证，连接到服务端 Worker。需要在 Master 上生成一个密钥对，包括一个公钥和一个私钥，然后将公钥复制到 Worker 上。当 Master 通过 ssh 连接 Worker 时，Worker 就会生成一个随机数，并用 Master 的公钥对随机数进行加密，发送给 Worker。Master 收到加密数之后，再用私钥进行解密，并将解密数回传给 Worker，Worker 确认解密数无误之后，允许 Master 进行连接。这就是一个公钥认证过程，其间不需要用户手工输入密码，主要过程是将 Master 节点公钥复制到 Worker 节点上[47]。

（1）在本地主机生成密钥对。

```
ssh-keygen -t rsa
```

这个命令生成一个密钥对：id_rsa（私钥文件）和 id_rsa.pub（公钥文件）。默认被保存在~/.ssh/目录下。

（2）将公钥添加到远程主机的 authorized_keys 文件中。

将文件上传到远程主机中：

```
scp ~/.ssh/id_rsa.pub  root@192.168.1.116:/root/
```

SSH 登录到远程主机 192.168.1.116，将公钥追加到 authorized_keys 文件中：

```
cat /root/id_rsa.pub >> /root/.ssh/authorized_keys
```

或直接运行命令：

```
cat ~/.ssh/id_dsa.pub|ssh root@192.168.1.116'sh -c "cat ->>~/.ssh/authorized_
keys"'
```

（3）重启 open-ssh 服务。

```
/etc/init.d/ssh restart
```

（4）本地测试。

```
ssh root@192.168.17.113
```

4. 安装 Hadoop

下面是 Hadoop 的安装过程和步骤：

（1）在官网地址 http://hadoop.apache.org 下载安装 Hadoop 2.7。

（2）配置 Hadoop 环境变量。

编辑 profile 文件。

```
vi /etc/profile
```

在 profile 文件中增加以下内容。

```
export JAVA_HOME=/usr/lib/jvm/jdk/
export HADOOP_INSTALL=/usr/local/hadoop
export PATH=$PATH：$HADOOP_INSTALL/bin
export PATH=$PATH：$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export YARN_HOME=$HADOOP_INSTALL
```

（3）编辑配置文件（根据设施环境配置相应文件）。

进入 Hadoop 所在目录/usr/local/hadoop/etc/hadoop。

配置 hadoop-env.sh 文件。

```
export JAVA_HOME=/usr/lib/jvm/jdk/
```

配置 core-site.xml 文件。

配置 yarn-site.xml 文件。

配置 mapred-site.xml 文件。

（4）创建 namenode 和 datanode 目录，并配置其相应路径。

创建 namenode 和 datanode 目录，执行以下命令：

```
$ mkdir /hdfs/namenode
$ mkdir /hdfs/datanode
```

执行命令后，再次回到目录/usr/local/hadoop/etc/hadoop，配置 hdfs-site.xml 文件，在文件中添加如下内容：

配置主节点名和端口号。

配置从节点名和端口号。

配置 datanode 的数据存储目录。

配置副本数。

（5）配置 Master 和 Slave 文件。

Master 文件负责配置主节点的主机名：

```
Master /*Master 为主节点主机名*/
```

配置 Slaves 文件添加从节点主机名，这样主节点就可以通过配置文件找到从节点，和从节点进行通信。

（6）将 Hadoop 的所有文件通过 pssh 分发到各个节点。

执行如下命令：

```
./pssh -h hosts.txt -r /hadoop nbsp; /
```

（7）格式化 Namenode（在 Hadoop 根目录下）。

```
./bin/hadoop namenode -format
```

（8）启动 Hadoop。

```
./sbin/start-all.sh
```

（9）查看是否配置和启动成功。

如果在 x86 机器上运行，则通过 jps 命令，查看相应的 JVM 进程。

5. 安装 Spark

进入官网 http://spark.apache.org/downloads.html，下载对应 Hadoop 版本的 Spark 程序包，Spark 下载页面如图 4-3 所示。

（1）下载 spark-2.0.0-bin-hadoop2.7.tgz，解压到/home/hadoop/spark 下。

```
1. cd /home/hadoop/spark
2. tar zxvf spark-2.0.0-bin-hadoop2.7.tgz
```

（2）修改/etc/profile。

```
1. # set spark environment
2. Export SPARK_HOME=/home/hadoop/spark/spark-2.0.0
3. Export PATH=$SPARK_HOME/bin
4. Export
5. SPARK_EXAMPLES_JAR=$SPARK_HOME/example/target/scala-2.11.1/spark-
examples_2.11.1-2.0
.0.jar
```

（3）配置 Spark。

修改配置文件 spark-env.sh。

```
1. cd /home/hadoop/spark/spark-1.6.1/conf
2. cp spark-env.sh.template  spark-env.sh
```

修改 spark-env.sh 如下：

```
1. # set scala environment
2. Export SCALA_HOME=/home/hadoop/scala/scala-2.11.1
3. Export JAVA_HOME=/home/javafile/jdk1.7.0_60
4. SPARK_MASTER_IP=mylinux
```

修改配置文件 slaves。

在 slaves 中添加你的 Slave 主机名：根据主机个数添加 Slave。

```
# A Spark worker will be started on each of the machines listed below.
# localhost
Test
```

配置 Slave。至此，Master 节点上的 Spark 已配置完毕。把 Master 上 Spark 相关配置复制到 Slave 中。注意，三台机器 Spark 所在目录必须一致，因为 Master 会登录到 Worker 上执行命令，Master 认为 Worker 的 Spark 路径与自己一样。

（4）启动 Spark。

```
cd /home/hadoop/spark/spark-1.6.1
sbin/start-all.sh
```

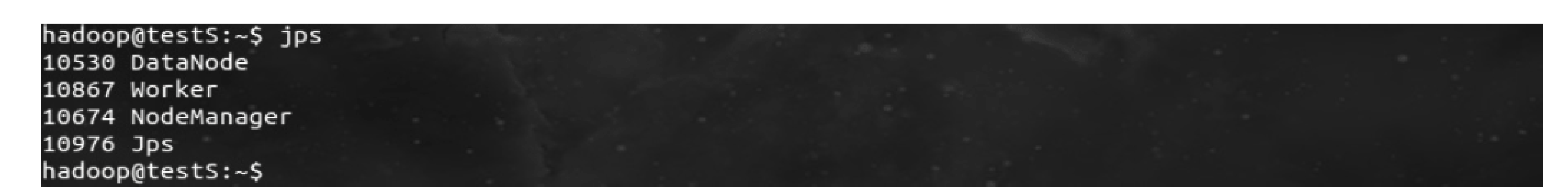
（5）验证 Spark 是否启动成功。

使用 JSP 命令，查看 Master 和 Slave 上的进程。

Master：

```
hadoop@mylinux:~$ jps
14454 Master
14124 ResourceManager
13948 SecondaryNameNode
13706 NameNode
15178 Jps
hadoop@mylinux:~$
```

Slave：



浏览器输入 http://masterIP:8080（将 Master 替换成你的 Master 主机名即可），也可以观察到集群的整个状态是否正常，如图 4-4 所示。集群会显示与图 4-4 类似的画面，masterIP 配置为用户的 Spark 集群的主节点 IP。

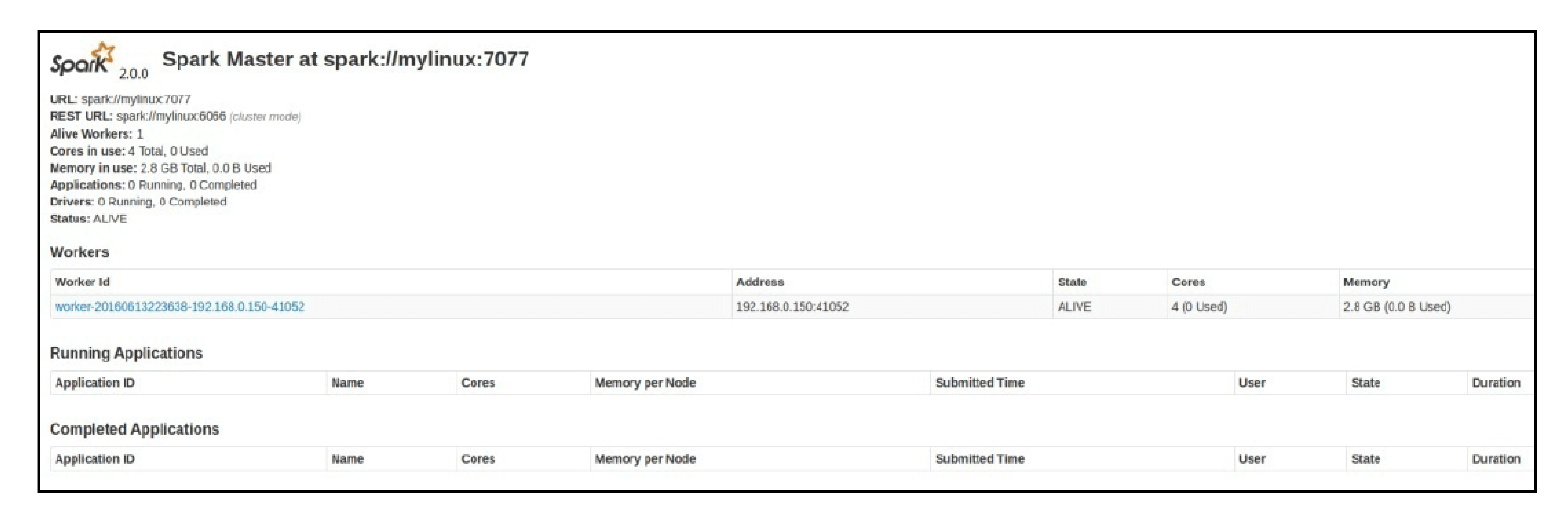


图 4-4　Spark Web 管理页

4.2.2　Spark Shell

Spark Shell 是 Spark 自带的一个快速原型开发工具，在 Spark 目录下面的 bin 目录下面。Spark 的交互式 Shell 提供了一个简单的方式来学习 Spark 的 API，同时也提供了强大的交互式数据处理能力。Spark Shell 支持 Scala 和 Python 两种语言[42]。

启动支持 Scala 的 Spark Shell 方式为./bin/spark-shell。

使用 yum 安装 Spark 之后，你可以直接在终端运行 spark-shell 命令，或者在 Spark 的 home 目录/usr/lib/spark 下运行 bin/spark-shell 命令，这样就可以进入到 Spark 命令行交互模式。

1. spark-shell 脚本代码

spark-shell 脚本代码如下：

```
#
# Shell script for starting the Spark Shell REPL

cygwin=false
case "`uname`" in
  CYGWIN*) cygwin=true;;
esac

# Enter posix mode for bash
set -o posix

## Global script variables
FWDIR="$(cd "`dirname "$0"`"/..; pwd)"

function usage() {
  echo "Usage: ./bin/spark-shell [options]"
"$FWDIR"/bin/spark-submit --help 2>&1 | grep -v Usage 1>&2
  exit 0
}

if [[ "$@" = *--help ]] || [[ "$@" = *-h ]]; then
  usage
fi

source "$FWDIR"/bin/utils.sh
SUBMIT_USAGE_FUNCTION=usage
gatherSparkSubmitOpts "$@"
```

```
# SPARK-4161: scala does not assume use of the java classpath,
# so we need to add the "-Dscala.usejavacp=true" flag mnually. We
# do this specifically for the Spark shell because the scala REPL
# has its own class loader, and any additional classpath specified
# through spark.driver.extraClassPath is not automatically propagated.
SPARK_SUBMIT_OPTS="$SPARK_SUBMIT_OPTS -Dscala.usejavacp=true"

function main() {
  if $cygwin; then
    # Workaround for issue involving JLine and Cygwin
    # (see http://sourceforge.net/p/jline/bugs/40/).
    # If you're using the Mintty terminal emulator in Cygwin, may need to set
the
    # "Backspace sends ^H" setting in "Keys" section of the Mintty options
    # (see https://github.com/sbt/sbt/issues/562).
    stty -icanon min 1 -echo > /dev/null 2>&1
    export SPARK_SUBMIT_OPTS="$SPARK_SUBMIT_OPTS -Djline.terminal=unix"
"$FWDIR"/bin/spark-submit --class org.apache.spark.repl.Main
"${SUBMISSION_OPTS[@]}" spark-shell "${APPLICATION_OPTS[@]}"
    stty icanon echo > /dev/null 2>&1
  else
    export SPARK_SUBMIT_OPTS
"$FWDIR"/bin/spark-submit --class org.apache.spark.repl.Main
"${SUBMISSION_OPTS[@]}" spark-shell "${APPLICATION_OPTS[@]}"
  fi
}

# Copy restore-TTY-on-exit functions from Scala script so spark-shell exits
properly even in
# binary distribution of Spark where Scala is not installed
exit_status=127
saved_stty=""

# restore stty settings (echo in particular)
function restoreSttySettings() {
  stty $saved_stty
  saved_stty=""
}
```

```
function onExit() {
  if [[ "$saved_stty" != "" ]]; then
    restoreSttySettings
  fi
  exit $exit_status
}

# to reenable echo if we are interrupted before completing.
trap onExit INT

# save terminal settings
saved_stty=$(stty -g 2>/dev/null)
# clear on error so we don't later try to restore them
if [[ ! $? ]]; then
  saved_stty=""
fi

main "$@"

# record the exit status lest it be overwritten:
# then reenable echo and propagate the code.
exit_status=$?
onExit
```

从上往下一步步分析，首先是判断是否为 Cygwin，这里用到了 bash 中的 case 语法：在 Linux 系统中， uname 命令的运行结果为 Linux，其值不等于 CYGWIN*，故 cygwin=false。

```
cygwin=false
case "`uname`" in
  CYGWIN*) cygwin=true;;
esac
```

开启 bash 的 posix 模式：

```
set -o posix
```

获取上级目录绝对路径用到了 dirname 命令，bash 中$0 是获取脚本名称：

```
FWDIR="$(cd "`dirname "$0"`"/..; pwd)"
```

从上面可以看到，其实最后调用的是 spark-submit 命令，并指定 --class 参数为 org.apache.spark.repl.Main 类，后面接的是 spark-submit 的提交参数，再后面是 spark-shell，最后是传递应用的参数。

最后，是获取 main 方法运行结果：

```
exit_status=$?
onExit
```

2. Spark-submit

完整的 Spark-submit 脚本内容如下：

```
# NOTE: Any changes in this file must be reflected in
SparkSubmitDriverBootstrapper.scala!

export SPARK_HOME="$(cd "`dirname "$0"`"/..; pwd)"
ORIG_ARGS=("$@")

# Set COLUMNS for progress bar
export COLUMNS=`tput cols`

while (($#)); do
  if [ "$1" = "--deploy-mode" ]; then
    SPARK_SUBMIT_DEPLOY_MODE=$2
  elif [ "$1" = "--properties-file" ]; then
    SPARK_SUBMIT_PROPERTIES_FILE=$2
  elif [ "$1" = "--driver-memory" ]; then
    export SPARK_SUBMIT_DRIVER_MEMORY=$2
  elif [ "$1" = "--driver-library-path" ]; then
    export SPARK_SUBMIT_LIBRARY_PATH=$2
  elif [ "$1" = "--driver-class-path" ]; then
    export SPARK_SUBMIT_CLASSPATH=$2
  elif [ "$1" = "--driver-java-options" ]; then
    export SPARK_SUBMIT_OPTS=$2
  elif [ "$1" = "--master" ]; then
    export MASTER=$2
  fi
  shift
done

if [ -z "$SPARK_CONF_DIR" ]; then
  export SPARK_CONF_DIR="$SPARK_HOME/conf"
fi
DEFAULT_PROPERTIES_FILE="$SPARK_CONF_DIR/spark-defaults.conf"
```

```
if [ "$MASTER" == "yarn-cluster" ]; then
  SPARK_SUBMIT_DEPLOY_MODE=cluster
fi
export SPARK_SUBMIT_DEPLOY_MODE=${SPARK_SUBMIT_DEPLOY_MODE:-"client"}
export SPARK_SUBMIT_PROPERTIES_FILE=${SPARK_SUBMIT_PROPERTIES_FILE:-
"$DEFAULT_PROPERTIES_FILE"}

# For client mode, the driver will be launched in the same JVM that launches
# SparkSubmit, so we may need to read the properties file for any extra class
# paths, library paths, java options and memory early on. Otherwise, it will
# be too late by the time the driver JVM has started.

if [[ "$SPARK_SUBMIT_DEPLOY_MODE" == "client"&& -f
"$SPARK_SUBMIT_PROPERTIES_FILE" ]]; then
  # Parse the properties file only if the special configs exist
  contains_special_configs=$(
    grep -e
"spark.driver.extra*\|spark.driver.memory""$SPARK_SUBMIT_PROPERTIES_FILE" | \
    grep -v "^[[:space:]]*#"
  )
  if [ -n "$contains_special_configs" ]; then
    export SPARK_SUBMIT_BOOTSTRAP_DRIVER=1
  fi
fi

exec "$SPARK_HOME"/bin/spark-class org.apache.spark.deploy.SparkSubmit
"${ORIG_ARGS[@]}"
```

首先是设置 SPARK_HOME，并保留原始输入参数：

```
export SPARK_HOME="$(cd "`dirname "$0"`"/..; pwd)"
ORIG_ARGS=("$@")
```

接下来，使用 while 语句配合 shift 命令，依次判断输入参数。设置 SPARK_CONF_DIR 变量，并判断 spark-submit 部署模式。如果 $SPARK_CONF_DIR/sparkdefaults.conf 文件存在，则检查是否设置 spark.driver.extra 开头的变量和 spark.driver.memory 变量，如果设置了，则 SPARK_SUBMIT_ BOOTSTRAP_DRIVER 设为 1。最后，执行的是 spark-class 命令，输入参数为 org.apache.spark. deploy. SparkSubmit 类名和原始参数。

3. Spark-class

该脚本首先还是判断是否是 Cygwin，并设置 SPARK_HOME 和 SPARK_CONF_DIR 变

量。运行 bin/load-spark-env.sh，加载 Spark 环境变量。spark-class 至少需要传递一个参数，如果没有，则会打印脚本使用说明 Usage: spark-class <class>[<args>]。如果设置了 SPARK_MEM 变量，则提示 SPARK_MEM 变量过时，应该使用 spark.executor.memory 或者 spark.driver.memory 变量。设置默认内存 DEFAULT_MEM 为 512MB，如果 SPARK_MEM 变量存在，则使用 SPARK_MEM 的值。

使用 case 语句判断 spark-class 传入的第一个参数的值：

```
SPARK_DAEMON_JAVA_OPTS="$SPARK_DAEMON_JAVA_OPTS -
Dspark.akka.logLifecycleEvents=true"

# Add java opts and memory settings for master, worker, history server,
executors, and repl.
case "$1" in
  # Master, Worker, and HistoryServer use SPARK_DAEMON_JAVA_OPTS (and specific
opts) + SPARK_DAEMON_MEMORY.
  'org.apache.spark.deploy.master.Master')
    OUR_JAVA_OPTS="$SPARK_DAEMON_JAVA_OPTS $SPARK_MASTER_OPTS"
    OUR_JAVA_MEM=${SPARK_DAEMON_MEMORY:-$DEFAULT_MEM}
    ;;
  'org.apache.spark.deploy.worker.Worker')
    OUR_JAVA_OPTS="$SPARK_DAEMON_JAVA_OPTS $SPARK_WORKER_OPTS"
    OUR_JAVA_MEM=${SPARK_DAEMON_MEMORY:-$DEFAULT_MEM}
    ;;
  'org.apache.spark.deploy.history.HistoryServer')
    OUR_JAVA_OPTS="$SPARK_DAEMON_JAVA_OPTS $SPARK_HISTORY_OPTS"
    OUR_JAVA_MEM=${SPARK_DAEMON_MEMORY:-$DEFAULT_MEM}
    ;;

  # Executors use SPARK_JAVA_OPTS + SPARK_EXECUTOR_MEMORY.
  'org.apache.spark.executor.CoarseGrainedExecutorBackend')
    OUR_JAVA_OPTS="$SPARK_JAVA_OPTS $SPARK_EXECUTOR_OPTS"
    OUR_JAVA_MEM=${SPARK_EXECUTOR_MEMORY:-$DEFAULT_MEM}
    ;;
  'org.apache.spark.executor.MesosExecutorBackend')
    OUR_JAVA_OPTS="$SPARK_JAVA_OPTS $SPARK_EXECUTOR_OPTS"
    OUR_JAVA_MEM=${SPARK_EXECUTOR_MEMORY:-$DEFAULT_MEM}
    export PYTHONPATH="$FWDIR/python:$PYTHONPATH"
    export PYTHONPATH="$FWDIR/python/lib/py4j-0.8.2.1-src.zip:$PYTHONPATH"
    ;;
```

```
 # Spark submit uses SPARK_JAVA_OPTS + SPARK_SUBMIT_OPTS +
 # SPARK_DRIVER_MEMORY + SPARK_SUBMIT_DRIVER_MEMORY.
 'org.apache.spark.deploy.SparkSubmit')
   OUR_JAVA_OPTS="$SPARK_JAVA_OPTS $SPARK_SUBMIT_OPTS"
   OUR_JAVA_MEM=${SPARK_DRIVER_MEMORY:-$DEFAULT_MEM}
   if [ -n "$SPARK_SUBMIT_LIBRARY_PATH" ]; then
     if [[ $OSTYPE == darwin* ]]; then
      export DYLD_LIBRARY_PATH="$SPARK_SUBMIT_LIBRARY_PATH:$DYLD_LIBRARY_PATH"
     else
      export LD_LIBRARY_PATH="$SPARK_SUBMIT_LIBRARY_PATH:$LD_LIBRARY_PATH"
     fi
   fi
   if [ -n "$SPARK_SUBMIT_DRIVER_MEMORY" ]; then
     OUR_JAVA_MEM="$SPARK_SUBMIT_DRIVER_MEMORY"
   fi
   ;;

 *)
   OUR_JAVA_OPTS="$SPARK_JAVA_OPTS"
   OUR_JAVA_MEM=${SPARK_DRIVER_MEMORY:-$DEFAULT_MEM}
   ;;
esac
```

可能存在以下几种情况：

```
org.apache.spark.deploy.master.Master
org.apache.spark.deploy.worker.Worker
org.apache.spark.deploy.history.HistoryServer
org.apache.spark.executor.CoarseGrainedExecutorBackend
org.apache.spark.executor.MesosExecutorBackend
org.apache.spark.deploy.SparkSubmit
```

判断 SPARK_SUBMIT_BOOTSTRAP_DRIVER 变量值，如果该值为 1，则运 org.apache.spark.deploy.SparkSubmitDriverBootstrapper 类，以替换原来的 org.apache.spark.deploy.SparkSubmit 的类，执行的脚本为 exec “$RUNNER” org.apache.spark.deploy.SparkSubmitDriverBootstrapper “$@”；否则，运行 java 命令 exec “$RUNNER” -cp “$CLASSPATH” $JAVA_OPTS “$@”。

从最后运行的脚本可以看到，spark-class 脚本的作用主要是查找 Java 命令、计算环境变量、设置 JAVA_OPTS 等，至于运行的是哪个 Java 类的 main 方法，取决于 SPARK_SUBMIT_BOOTSTRAP_DRIVER 变量的值。

4.2.3 Spark RDD

Spark 最重要的一个抽象概念是弹性分布式数据集（Resilient Distributed Dataset），英文简称 RDD。RDD 有两种创建方式：

- 从 Hadoop 的文件系统输入（例如 HDFS）。
- 由其他已存在的 RDD 转换得到新的 RDD。

RDD 有两种类型的操作，分别是 Action（返回 values）和 Transformations（返回一个新的 RDD）。

下面的例子是通过加载 Spark 目录下的 README.md 文件生成 RDD 的例子：

```
scala> val textFile = sc.textFile("README.md") textFile: spark.RDD[String] =
spark.MappedRDD@2ee9
b6e3
```

actions 示例如下：

```
scala> textFile.count() // Number of items in this RDD res0: Long = 126 scala>
textFile.first() // First item in this RDD res1: String = # Apache Spark
```

下面是 transformations 示例，使用 filter 操作返回了一个新的 RDD，该 RDD 为文件中数据项的子集，该子集符合过滤条件：

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark: spark.RDD[Stri-
ng] = spark.FilteredRDD@7dd4af09
```

Spark 也支持将 actions 和 transformations 一起使用：

```
scala> textFile.filter(line => line.contains("Spark")).count() // How many
lines contain "Spark"? res3: Long = 15
```

获取 RDD 的途径有以下几种：

（1）从共享的文件系统获取（如：HDFS）。
（2）通过已存在的 RDD 转换。
（3）将已存在 Scala 集合（只要是 Seq 对象）并行化，通过调用 SparkContext 的 parallelize 方法实现。
（4）改变现有 RDD 的持久性，RDD 是懒散、短暂的。

RDD 的固化有两种方法：

- cache：缓存至内存。
- save：保存到分布式文件系统。

操作 RDD 的两个动作：

- Actions：对数据集计算后返回一个数值 value 给驱动程序，例如：Reduce 将数据集的所有元素用某个函数聚合后，将最终结果返回给程序。Actions 具体内容如表 4-3 所示。

表 4-3 Actions 具体内容

函数	描述
reduce(func)	通过函数 func 聚集数据集中的所有元素。Func 函数接受 2 个参数，返回一个值。这个函数必须是关联性的，确保可以被正确地并发执行
collect()	在 Driver 的程序中，以数组的形式，返回数据集的所有元素。这通常会在使用 filter 或者其他操作后，返回一个足够小的数据子集再使用，直接将整个 RDD 集 Collect 返回，很可能会让 Driver 程序 OOM
count()	返回数据集的元素个数
take(n)	返回一个数组，由数据集的前 n 个元素组成。注意，这个操作目前并非在多个节点上并行执行，而是 Driver 程序所在的机器，单机计算所有的元素（Gateway 的内存压力会增大，需要谨慎使用）
first()	返回数据集的第一个元素，类似于 take（1）
saveAsTextFile(path)	将数据集的元素，以 textfile 的形式保存到本地文件系统、HDFS 或者任何其他 Hadoop 支持的文件系统。Spark 将会调用每个元素的 toString 方法，并将它转换为文件中的一行文本
saveAsSequenceFile(path)	将数据集的元素，以 sequencefile 的格式，保存到指定的目录下、本地系统、HDFS 或者任何其他 Hadoop 支持的文件系统。RDD 的元素必须由 key-value 对组成，并都实现了 Hadoop 的 Writable 接口，或隐式可以转换为 Writable（Spark 包括了基本类型的转换，例如 Int、Double、String 等）
foreach(func)	在数据集的每一个元素上，运行函数 func。这通常用于更新一个累加器变量，或者和外部存储系统做交互

- Transformation：根据数据集创建一个新的数据集，计算后返回一个新 RDD，如：Map 将数据的每个元素经过某个函数计算后，返回一个新的分布式数据集。Transformation 具体内容如表 4-4 所示。

表 4-4 Transformation 具体内容

函数	描述
map(func)	返回一个新的分布式数据集，由每个原元素经过 func 函数转换后组成
filter(func)	返回一个新的数据集，由经过 func 函数后返回值为 true 的原元素组成
flatMap(func)	类似于 map，但是每一个输入元素，会被映射为 0 到多个输出元素（因此，func 函数的返回值是一个 Seq，而不是单一元素）
sample(withReplacement, frac, seed)	根据给定的随机种子 seed，随机抽样出数量为 frac 的数据
union(otherDataset)	返回一个新的数据集，由原数据集和参数联合而成
groupByKey([numTasks])	在一个由（K,V）对组成的数据集上调用，返回一个（K，Seq[V])对的数据集。注意：默认情况下，使用 8 个并行任务进行分组，你可以传入 numTask 可选参数，根据数据量设置不同数目的 Task

（续表）

函数	描述
reduceByKey(func, [numTasks])	在一个（K，V)对的数据集上使用，返回一个（K，V）对的数据集，key 相同的值，都被使用指定的 reduce 函数聚合到一起。和 groupbykey 类似，任务的个数是可以通过第二个可选参数来配置的
join(otherDataset, [numTasks])	在类型为（K,V）和（K,W）的数据集上调用，返回一个（K,(V,W))对，每个 key 中的所有元素都在一起的数据集
groupWith(otherDataset, [numTasks])	在类型为（K,V）和（K,W）的数据集上调用，返回一个数据集，组成元素为（K, Seq[V], Seq[W]) Tuples。这个操作在其他框架，称为 CoGroup
cartesian(otherDataset)	笛卡尔积。但在数据集 T 和 U 上调用时，返回一个（T，U）对的数据集，所有元素交互进行笛卡尔积
flatMap(func)	类似于 map，但是每一个输入元素，会被映射为 0 到多个输出元素（因此，func 函数的返回值是一个 Seq，而不是单一元素）

RDD 内部的设计，每个 RDD 都需要包含以下 4 个部分：

（1）源数据分割后的数据块，源代码中的 splits 变量。

（2）关于“血统”的信息，源码中的 dependencies 变量。

（3）一个计算函数（该 RDD 如何通过父 RDD 计算得到），源码中的 iterator(split)和 compute 函数。

（4）一些关于如何分块和数据存放位置的元信息，如源码中的 partitioner 和 preferredLocations。例如：一个从分布式文件系统中的文件得到的 RDD 具有的数据块是通过切分各个文件得到的，它是没有父 RDD 的，它的计算函数只是读取文件的每一行并作为一个元素返回给 RDD。

对于一个通过 map 函数得到的 RDD，它会具有和父 RDD 相同的数据块，它的计算函数是对每个父 RDD 中的元素所执行的一个函数 。

RDD 根据 useDisk、useMemory、deserialized、replication 4 个参数的组合提供了 11 种存储级别：

（1）val NONE = new StorageLevel(false, false, false)

（2）val DISK_ONLY = new StorageLevel(true, false, false)

（3）val DISK_ONLY_2 = new StorageLevel(true, false, false, 2)

（4）val MEMORY_ONLY = new StorageLevel(false, true, true)

（5）val MEMORY_ONLY_2 = new StorageLevel(false, true, true, 2)

（6）val MEMORY_ONLY_SER = new StorageLevel(false, true, false)

（7）val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, 2)

（8）val MEMORY_AND_DISK = new StorageLevel(true, true, true)

（9）val MEMORY_AND_DISK_2 = new StorageLevel(true, true, true, 2)

（10）val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false)

（11）val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, 2)

4.2.4　Shark（Hive on Spark 大型的数据仓库系统）

Shark 是 UC Berkeley AMPLAB 开源的一款数据仓库产品，它完全兼容 Hive 的 HQL 语法，但与 Hive 不同的是，Hive 的计算框架采用 Map-Reduce，而 Shark 采用 Spark。所以，Hive 是 SQL on Map-Reduce，而 Shark 是 Hive on Spark。其架构如图 4-5 所示。

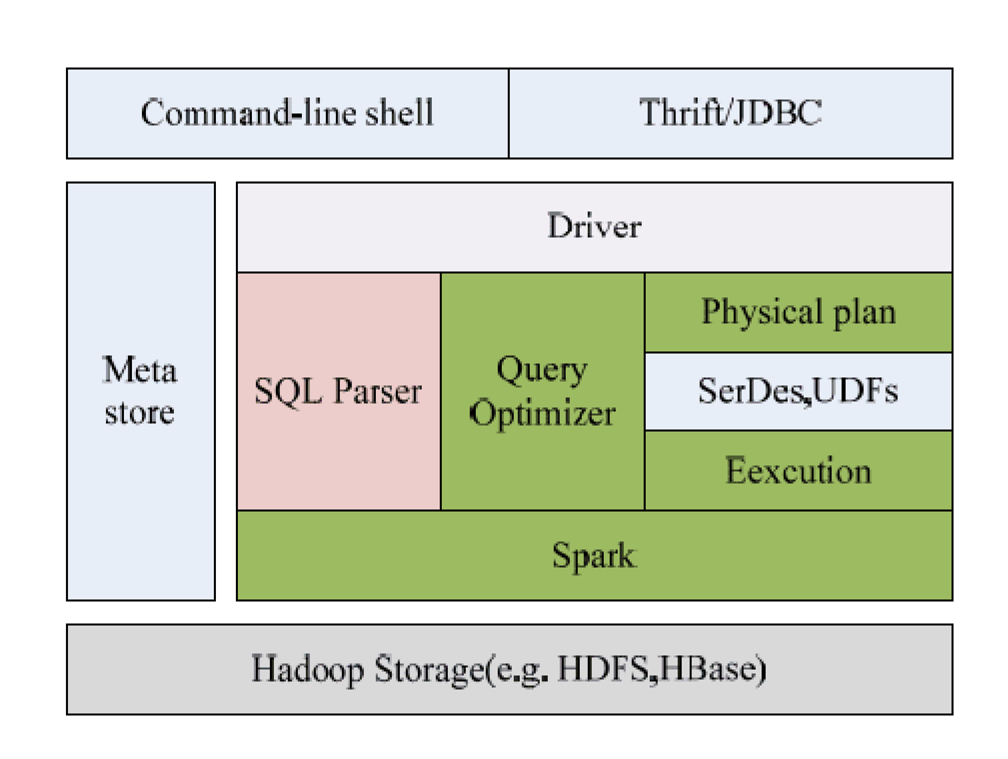


图 4-5　Shark 架构

为了最大限度地保持与 Hive 的兼容性，Shark 复用了 Hive 的大部分组件，如下所示：

（1）SQL Parser & Plan generation：Shark 完全兼容 Hive 的 HQL 语法，而且 Shark 使用了 Hive 的 API 来实现 query Parsing 和 query Plan generation，仅仅最后的 Physical Plan execution 阶段用 Spark 代替 Hadoop Map-Reduce。

（2）Metastore：Shark 采用和 Hive 一样的 meta 信息，Hive 里创建的表用 Shark 可无缝访问。

（3）SerDe：Shark 的序列化机制以及数据类型与 Hive 完全一致。

（4）UDF：Shark 可重用 Hive 里的所有 UDF。通过配置 Shark 参数，Shark 可以自动在内存中缓存特定的 RDD，实现数据重用，进而加快特定数据集的检索。同时，Shark 通过 UDF 用户自定义函数实现特定的数据分析学习算法，使得 SQL 数据查询和运算分析能结合在一起，最大化 RDD 的重复使用。

（5）Driver：Shark 在 Hive 的 CliDriver 基础上进行了一个封装，生成一个 SharkCliDriver，这是 shark 命令的入口。

（6）ThriftServer：Shark 在 Hive 的 ThriftServer（支持 JDBC/ODBC）基础上，做了一个封装，生成了一个 SharkServer，也提供 JDBC/ODBC 服务。

Shark 即 Hive on Spark，本质上是通过 Hive 的 HQL 解析，把 HQL 翻译成 Spark 上的 RDD 操作，然后通过 Hive 的 metadata 获取数据库里的表信息，实际 HDFS 上的数据和文件，会由 Shark 获取并放到 Spark 上运算。Shark 的特点就是快，完全兼容 Hive，且可以在 shell 模式下使用 rdd2sql()这样的 API，把 HQL 得到的结果集，继续在 Scala 环境下运算，支持自己编写简单的机器学习或简单的分析处理函数，以对 HQL 结果进一步分析计算。

4.3 Spark 配置

Spark 主要提供三种位置配置系统：

（1）环境变量：用来启动 Spark workers，可以设置在你的驱动程序或者 conf/spark-env.sh 脚本中。

（2）Java 系统性能：可以控制内部的配置参数，有两种设置方法：

- 编程的方式（程序中在创建 SparkContext 之前，使用 System.setProperty（“xx”，“xxx”）语句设置相应系统属性值）。
- 在 conf/spark-env.sh 中配置环境变量 SPARK_JAVA_OPTS。

（3）日志配置：通过 log4j.properties 实现。

4.3.1 环境变量

Spark 安装目录下的 conf/spark-env.sh 脚本决定了如何初始化 Worker Nodes 的 JVM，甚至决定了你在本地如何运行 spark-shell。在 Git 库中这个脚本默认是不存在的，但是你可以自己创建它并通过复制 con/spark-env.sh.template 中的内容来配置，最后要确保你创建的文件可执行。

在 spark-env.sh 中你至少有两个变量要设置：

（1）SCALA_HOM：指向你的 Scala 安装路径；或者是 SCALA_LIBRARY_PATH 指向 Scala library JARS 所在的目录。如果你是通过 DEB 或者 RPM 安装的 Scala，他们是没有 SCALA_HOME 的，但是他们的 libraries 是分离的，默认在/usr/share/java 中查找 scala-library.jar。

（2）MESOS_NATIVE_LIBRAR：如果你要在 Mesos 上运行集群的话，必须设置它。

另外，还有其他 4 个变量来控制执行。应该将他们设置在启动驱动程序的环境中，以取代设置在 spark-env.sh 文件中，因为这些设置可以自动传递给 Workers。将他们设置在每个作业中而不是 spark-env.sh 中，这样确保了每个作业有他们自己的配置。

- SPARK_JAVA_ORTS：添加 JVM 选项。可以通过-D 来获取任何系统属性。
- SPARK_CLASS_PATH：添加元素到 Spark 的 classpth 中。
- SPARK_LIBARAT_OATH：添加本地 libraries 的查找目录。
- SPARK_MEM：设置每个节点所能使用的内存总量。他们应该和 JVM‘s -Xmx 选项的格式保持一致（e.g.300m 或 1g）。注意：这个选项将很快被弃用，转而支持系统属性 spark.executor.memory，所以我们不推荐将它使用在新代码中。

如果将他们设置在 spark-env.sh 中，他们将覆盖用户程序设定的值，这是不可取的。如果环境允许，你可以选择在 spark-env.sh 中设置他们，仅当用户程序没有做任何设置时，例如：

```
if [ -z "$SPARK_JAVA_OPTS" ] ; then
```

```
SPARK_JAVA_OPTS="-verbose:gc"
fi
```

4.3.2　系统属性

通过设置系统属性来配置 Spark，你必须通过以下两种方式中的任意一个来达到目的：

- 在 JVM 中通过-D 标志（例如：java -Dspark.cores.max=5 MyProgram）。
- 在你的程序中创建 SparkContext 之前调用 System.setProperty，如下所示：

```
 System.setProperty("spark.cores.max", "5")
val sc = new SparkContext(...)
```

更多可配置的控制内部设置的系统属性已经有了合理的默认属性值。然而，有 5 个属性通常是你想要去控制的，如表 4-5 所示。

表 4-5　控制内部设置的系统属性（1）

属性名称	默认值	含义
spark.executor.memory	512m	每个处理器可以使用的内存大小，跟 JVM 的内存表示的字符串格式是一样的，比如：'512m'，'2g'
spark.serializer	spark.JavaSerializer	一个类名，用于序列化网络传输或者以序列化形式缓存起来的各种对象。默认情况下 Java 的序列化机制可以序列化任何实现了 Serializable 接口的对象，但是速度是很慢的，因此当你在意运行速度的时候我们建议你使用 spark.KryoSerializer 并且配置 Kryo serialization。可以是任何 spark.Serializer 的子类
spark.kryo.registrator	(none)	如果你使用的是 Kryo 序列化，就要为 Kryo 设置这个类去注册你自定义的类。这个类需要继承 spark.KryoRegistrator。可以参考调优指南获取更多的信息
spark.local.dir	/tmp	设置 Spark 的暂存目录，包括映射输出文件盒需要存储在磁盘上的 RDDs。这个磁盘目录在你的系统上面访问速度越快越好。可以用逗号隔开来设置多个目录
spark.cores.max	(infinite)	当运行在一个独立部署集群上或者是一个粗粒度共享模式的 Mesos 集群上的时候，最多可以请求多少个 CPU 核心。默认是所有的都能用

除了表 4-5 这些属性，在某些情况下表 4-6 所示的属性可能也是需要设置的。

表 4-6　控制内部设置的系统属性（2）

属性名	默认值	含义
spark.mesos.coarse	false	如果设置为了“true”，将以粗粒度共享模式运行在 Mesos 集群上，这时候 Spark 会在每台机器上面获得一个长期运行的 Mesos 任务，而不是对每个 Spark 任务都要产生一个 Mesos 任务。对于很多短查询，这个可能会有些许的延迟，但是会大大提高 Spark 工作时的资源利用率
spark.default.parallelism	8	在用户没有指定时，用于分布式随机操作（groupByKey、reduceByKey 等）的默认的任务数

（续表）

属性名	默认值	含义
spark.storage.memory Fraction	0.66	Spark 用于缓存的内存大小所占用的 Java 堆的比率。这个不应该大于 JVM 中老年代所分配的内存大小，默认情况下老年代大小是堆大小的 2/3，但是你可以通过配置你的老年代的大小，然后再去增加这个比率
spark.ui.port	(random)	你的应用程序控制面板端口号，控制面板中可以显示每个 RDD 的内存使用情况
spark.shuffle.compress	true	是否压缩映射输出文件，通常设置为 true 是个不错的选择
spark.broadcast. compress	true	广播变量在发送之前是否先要被压缩，通常设置为 true 是个不错的选择
spark.rdd.compress	false	是否要压缩序列化的 RDD 分区（比如，StorageLevel.MEMORY_ONLY_SER）。在消耗一点额外的 CPU 时间的代价下，可以极大地减少空间的使用
spark.reducer.maxMbIn Flight	48	同时获取每一个分解任务的时候，映射输出文件的最大的尺寸（以兆为单位）。由于对每个输出都需要我们去创建一个缓冲区去接受它，这个属性值代表了对每个分解任务所使用的内存的一个上限值，因此除非你机器内存很大，最好还是配置一下这个值
spark.closure.serializer	spark.Java Serializer	用于闭包的序列化类。通常 Java 是可以胜任的，除非在你的驱动程序中分布式函数（比如 map 函数）引用了大量的对象
spark.kryoserializer. buffer.mb	32	Kryo 中运行的对象的最大尺寸（Kryo 库需要创建一个不小于最大的单个序列化对象的缓存区）。如果在 Kryo 中出现“buffer limit exceeded”异常，你就需要去增加这个值了。注意，对每个 Worker 而言，一个核心就会有一个缓冲
spark.broadcast.factory	spark.broad cast.Http Broadcast Factory	使用哪一个广播实现
spark.locality.wait	3000	在发布一个本地数据任务的时候，放弃并发布到一个非本地数据的地方前，需要等待的时间。如果你的很多任务都是长时间运行的任务，并且看到了很多的脏数据的话，你就该增加这个值了。但是一般情况下默认值就可以很好地工作了
spark.worker.timeout	60	如果超过这个时间，独立部署 Master 还没有收到 Worker 的心跳回复，那么就认为这个 Worker 已经丢失了
spark.akka.frameSize	10	在控制面板通信（序列化任务和任务结果）的时候消息尺寸的最大值，单位是 MB。如果你需要给驱动器发回大尺寸的结果（比如使用在一个大的数据集上面使用 collect()方法），那么你就该增加这个值了
spark.akka.threads	4	用于通信的 actor 线程数量。如果驱动器有很多 CPU 核心，那么在大集群上可以增大这个值
spark.akka.timeout	20	Spark 节点之间通信的超时时间，以秒为单位
spark.driver.host	(local hostname)	驱动器监听主机名或者 IP 地址
spark.driver.port	(random)	驱动器监听端口号

（续表）

属性名	默认值	含义
spark.cleaner.ttl	(disable)	Spark 记忆任何元数据续表（stages 生成、任务生成等）的时间（秒）。周期性清除保证在这个时间之前的元数据会被遗忘。当长时间几小时、几天地运行 Spark 的时候设置这个是很有用的。注意：任何内存中的 RDD 只要过了这个时间就会被清除掉
spark.streaming.block Interval	200	从网络中批量接受对象时的持续时间
spark.task.maxFailures	4	task 失败重试次数

4.3.3　配置日志

Spark 使用 log4j 来记录。你可以在 conf 目录中添加 log4j.properties 文件来配置。一种方法是复制本地已存在的 log4j.properties.template。

4.3.4　Spark 硬件配置

针对 Spark 部署和开发的一个共同的问题是如何配置硬件，而正确的硬件将取决于环境。

1. 存储系统

Spark 任务需要从一些外部的存储系统加载数据（如：HDFS 或者 HBase），重要的是存储系统要接近 Spark 系统：

（1）如果可能，运行 Spark 在相同的 HDFS 节点，最简单的方法是建立一个引发相同的节点上的集群独立模式（http://spark.apache.org/docs/latest/spark-standalone.html），配置 Spark 的 Configure 和 Hadoop 的内存、CPU 使用避免干扰（对于 Hadoop），或者你能够运行 Hadoop 和 Spark 在一个相同的 cluster manager，像 Mesos、Hadoop YARN。

（2）如果可能，运行 Spark 在不同的节点上，需要使用相同局域网内部的 HDFS 节点。

（3）对于低延迟数据存储如同 HBase，使用不同的节点上的数据比使用本地存储系统数据干扰更小（但是 HBase 存储比本地存储在避免干扰性方面表现得更好）。

2. 本地硬盘

虽然 Spark 能够在内存中执行大量的计算，它仍然需要本地硬盘作为数据的存储，不适合把数据存储在 RAM 中，以及保护中间的输出阶段，我们推荐每个节点有 4~8 个硬盘。如果没有配置 RAID（就如同不同的挂载点），那就需要在 Linux 中挂载硬盘使用 noatime option（http://www.centos.org/docs/5/html/Global_File_System/s2-manage-mountnoatime.html）以减少不必要的写操作。在 Spark 里面，配置 spark.local.dir 变量以一个“，”号隔开（http://spark.apache.org/docs/latest/config-uration.html），如果你正在运行着 HDFS，它正好和 HDFS 放在一个相同的硬盘上。

3. 内存

一般而言，Spark 能够运行在任意的 8GB~几百 GB 内存的机器上，所有情况下，推荐最多给 Spark 配置 75%的内存容量，其他的容量供系统和 buffer 缓存使用。

内存需要多大是依靠 Application 决定的，确定应用使用多少内存特定大小，需要加载一部分特定的数据到 Spark RDD 并使用 UI 的存储选项卡（http://<driver-node>:4040）观测内存使用量。注意，内存使用量大大影响存储水平和序列化格式，可以通过这个地址查看优化方法（http://spark.apache.org/docs/latest/tuning.html ）。

最后，注意 Java VM 在超过 200 GB 的 RAM 上并不总是表现良好。如果这样的 RAM 机器，可以在上面多跑几个 Worker，在 Spark 的独立模式中，能够在每个节点上设置多个 Workers ，设置 conf/spark-env.sh 中的 SPARK_WORKER_INSTANCES 变量，并且设置 SPARK_WORKER_CORES 的核数。

4. 网络

根据经验，当数据在内存中，使用万兆网卡程序将运行得更快，特别是“distributed reduce application”当中，使用了 group-bys、reduce-bys 和 SQL 的 join 的操作的时候。在一个任何给定的 application 中，你能够通过 UI 查看 Spark 的 shuffles 的过程及多大的数据执行 shuffles。

5. CPU 核数

Spark 每个集群要启动成千上万的线程，每个集群的核数至少是 8~16 核。工作负载是依靠 CPU ，所以需要更多的 CPU 处理能力，一旦数据放在内存中，运行更多的应用取决于 CPU 或者带宽。

4.4 Spark 模式部署概述

本节简单介绍一下 Spark 如何在集群上运行，以使读者更易理解其中的组件。

1. 组件

Spark 应用在集群上以独立的进程集合运行，在你的主程序（称作驱动程序）中以 SparkContext 对象来调节。特别地，为了在集群上运行，SparkContext 可以与几个类型的集群管理器（Spark 自身单独的集群管理器或者 Mesos/YARN）相连接，这些集群管理器可以在应用间分配资源。一旦连接，Spark 需要在集群上的线程池子节点，也就是那些执行计算和存储应用数据的工作进程。然后，它将把你的应用代码（以 JAR 或者 Python 定义的文件并传送到 SparkContext）发送到线程池。最后，SparkContext 发送任务让线程池运行，如图 4-6 所示。

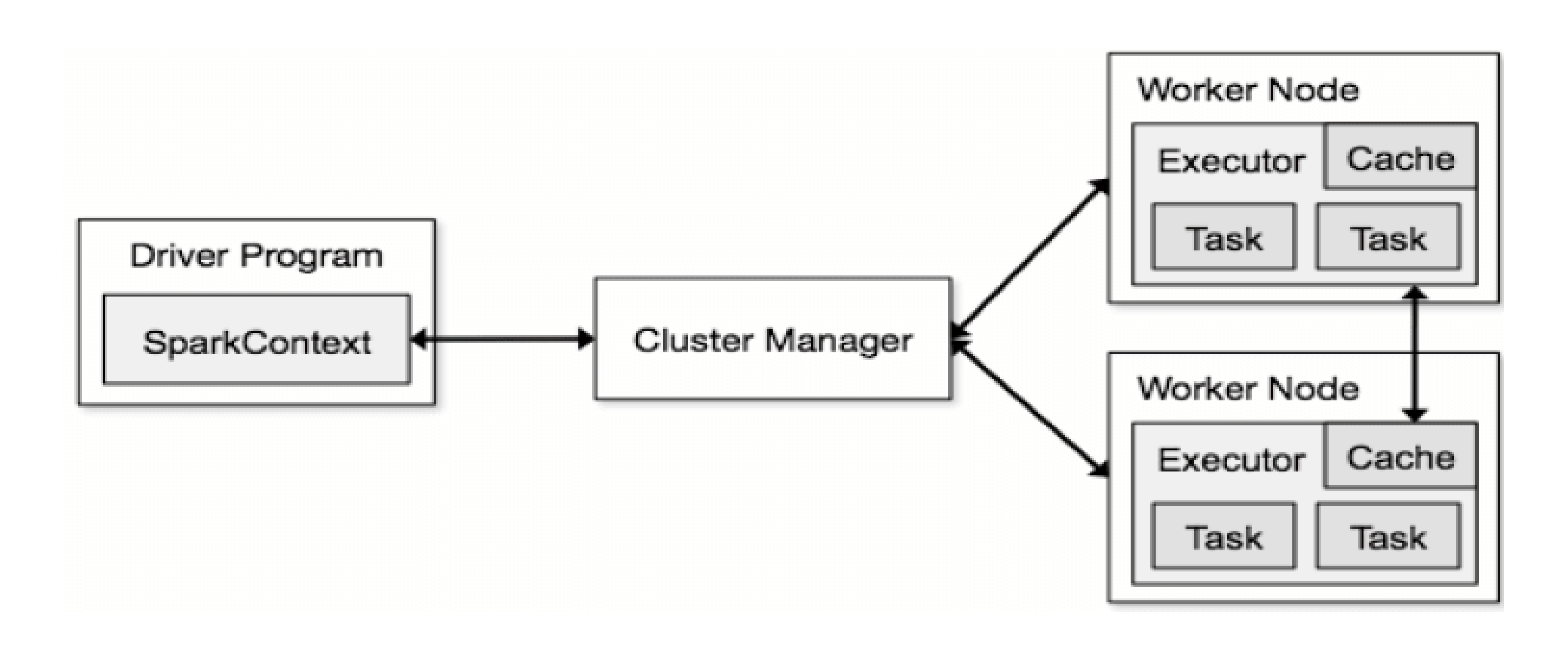


图 4-6　Spark 集群进程

关于这个架构有几个有用的地方需要注意：

（1）各个应用有自己的线程池进程，会在整个应用的运行过程中保持并在多个线程中运行任务。这样做的好处是把应用相互孤立，既在调度方面（各个驱动调度它自己的任务）也在执行方面（不同应用的任务在不同的 JVM 上运行）。然而，这也意味着若不把数据写到额外的存储系统的话，数据就无法在不同的 Spark 应用间（SparkContext 的实例）共享。

（2）对于潜在的集群管理器来说，Spark 是不可知的。只要它需要线程池的进程和它们间的通信，那么即使是在支持其他应用的集群管理器（例如，Mesos/YARN）上运行也相对简单。

（3）因为在集群上驱动调度任务，它应该运行最接近的工作节点，在相同的局域网内更好。如果你想对远程的集群发送请求，较好的选择是为驱动打开一个 RPC，让它就近提交操作，而不是运行离工作节点很远的驱动。

2. 集群管理类型

系统目前支持 3 种集群管理：

- 单例模式：一种简单的集群管理，其包括一个很容易搭建集群的 Spark。
- Apache Mesos 模式：一种通用的集群管理，可以运行 Hadoop MapReduce 和服务应用的模式。
- Hadoop YARN 模式：Hadoop2.0 中的资源管理模式。

其实，在 Amazon EC2（亚马逊弹性计算云）中 Spark 的 EC2 启动脚本可以很容易地启动单例模式。

3. 给集群发布代码

给集群发布代码的一种推荐的方式是通过 SparkContext 的构造器，这个构造器可以给工作节点生成（Java/Scala）JAR 文件列表或者（Python）.egg 文件和.zip 包文件。你也可以执行 SparkContext.addJar 和 addFile 来动态地创建发送文件。

4. 监控器

每个驱动程序有一个 Web UI，典型的是在 4040 端口，你可以看到有关运行的任务、程序和存储空间大小等信息。你可以在浏览器中输入简单的 URL 方式来访问 Web UI（http://<

驱动节点>:4040）。监控器也可以 指导描述其他监控器信息。

5. 调度

Spark 可以通过在应用外（集群管理水平）和应用内（如果在同一个 SparkContext 中有多个计算指令）进行资源分配。

6. 术语表

Spark 集群术语如表 4-7 所示。

表 4-7　Spark 集群术语

术语	意思
Application	在 Spark 上构建用户程序，　由驱动程序在集群上执行
Drive program	运行 man 函数的进程，同时也创建 SparkContex
Cluster manager	获取集群上资源的扩展服务（例如：单例模式管理员、Mesos、YARN）
Workenode	在集群中可以运行应用的任何节点
Executor	在工作员节点中为应用所启动的一个进程，以便运行任务以及可以在内存或是硬盘中保存数据。每一个应用都拥有自己的执行者
Task	一个可以给执行者发送数据的工作单元
Job	一个由从 Spark 动作中获得回应的多任务组成的并行计算（例如：保存、收集），你可以在驱动日志中看到这个术语
Stage	每个工作被分为很多小的任务集合互称为阶段（与 MapReduce 中的 map 和 reduce 阶段相似）， 你可以在驱动日志中看到这个术语

4.5 Spark Streaming 实时计算框架

随着大数据的发展，人们对大数据的处理要求也越来越高，原有的批处理框架 MapReduce 适合离线计算，却无法满足实时性要求较高的业务，如实时推荐、用户行为分析等[43]。Spark Streaming 是建立在 Spark 上的实时计算框架，通过它提供的丰富的 API、基于内存的高速执行引擎，用户可以结合流式、批处理和交互试查询应用。本节将介绍 Spark Streaming 实时计算框架的原理、特点与适用场景[44,45]。

Spark 是一个类似于 MapReduce 的分布式计算框架，其核心是弹性分布式数据集，它提供了比 MapReduce 更丰富的模型，可以在快速内存中对数据集进行多次迭代，以支持复杂的数据挖掘算法和图形计算算法[46,48]。Spark Streaming 是一种构建在 Spark 上的实时计算框架，它扩展了 Spark 处理大规模流式数据的能力。

Spark Streaming 的优势在于：

- 能运行在 100+的节点上，并达到秒级延迟。
- 使用基于内存的 Spark 作为执行引擎，具有高效和容错的特性。
- 能集成 Spark 的批处理和交互查询。
- 为实现复杂的算法提供与批处理类似的简单接口。

Spark Streaming 的基本原理是将输入数据流以时间片（秒级）为单位进行拆分，然后以类似批处理的方式处理每个时间片数据，其基本原理如图 4-7 所示。首先，Spark Streaming 把实时输入数据流以时间片Δt（如 1 秒）为单位切分成块。Spark Streaming 会把每块数据作为一个 RDD，并使用 RDD 操作处理每一小块数据。每个块都会生成一个 Spark Job 处理，最终结果也返回多块。

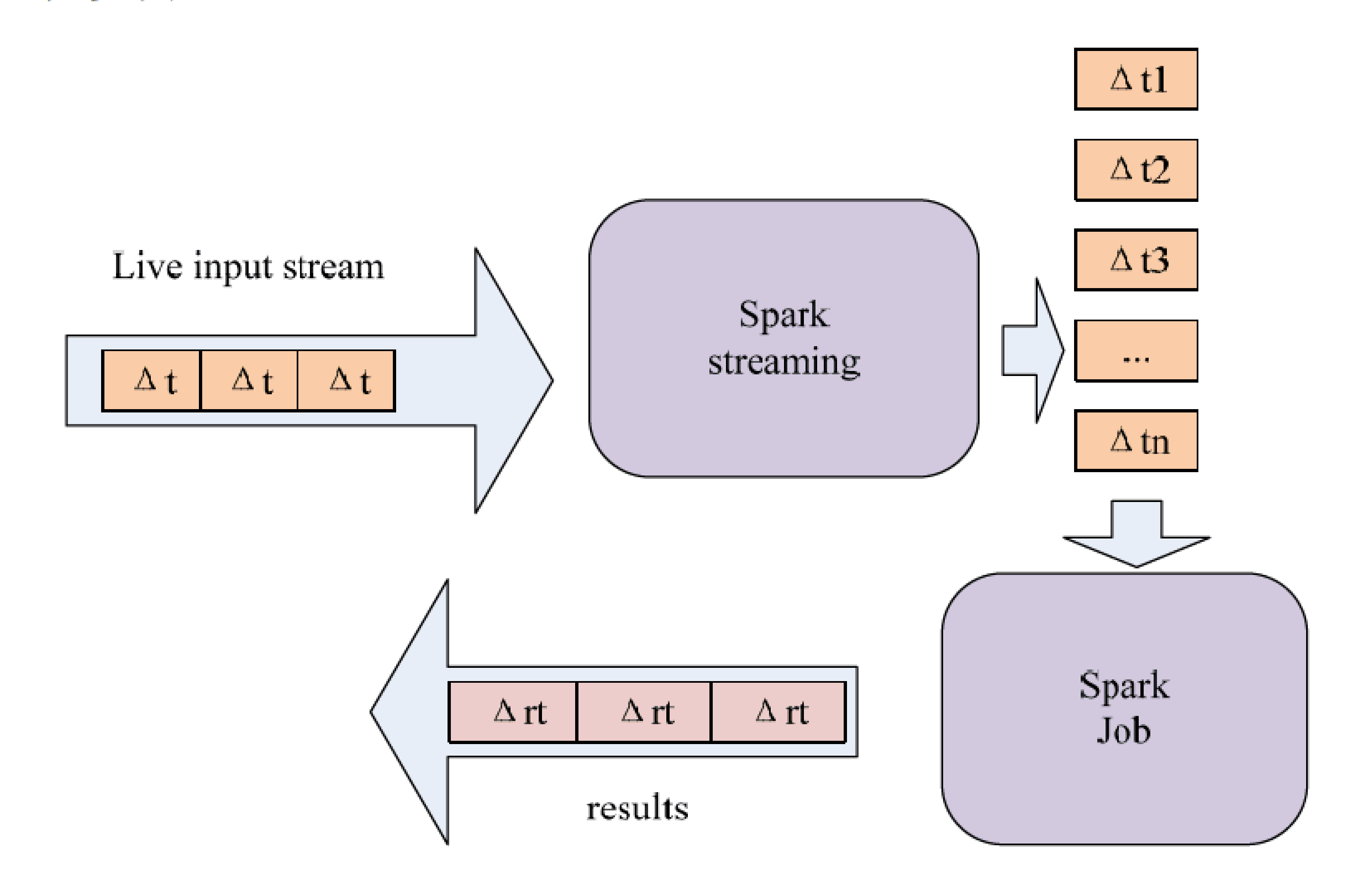


图 4-7　Spark Streaming 基本原理

下面介绍 Spark Streaming 内部实现原理。使用 Spark Streaming 编写的程序与编写 Spark 程序非常相似，在 Spark 程序中，主要通过操作 RDD（Resilient Distributed Datasets，弹性分布式数据集）提供的接口（如 map、reduce、filter 等）实现数据的批处理；而在 Spark Streaming 中，则通过操作 DStream（表示数据流的 RDD 序列）提供的接口实现数据的批处理，这些接口和 RDD 提供的接口类似。图 4-8 展示了由 Spark Streaming 程序到 Spark jobs 的转换图。

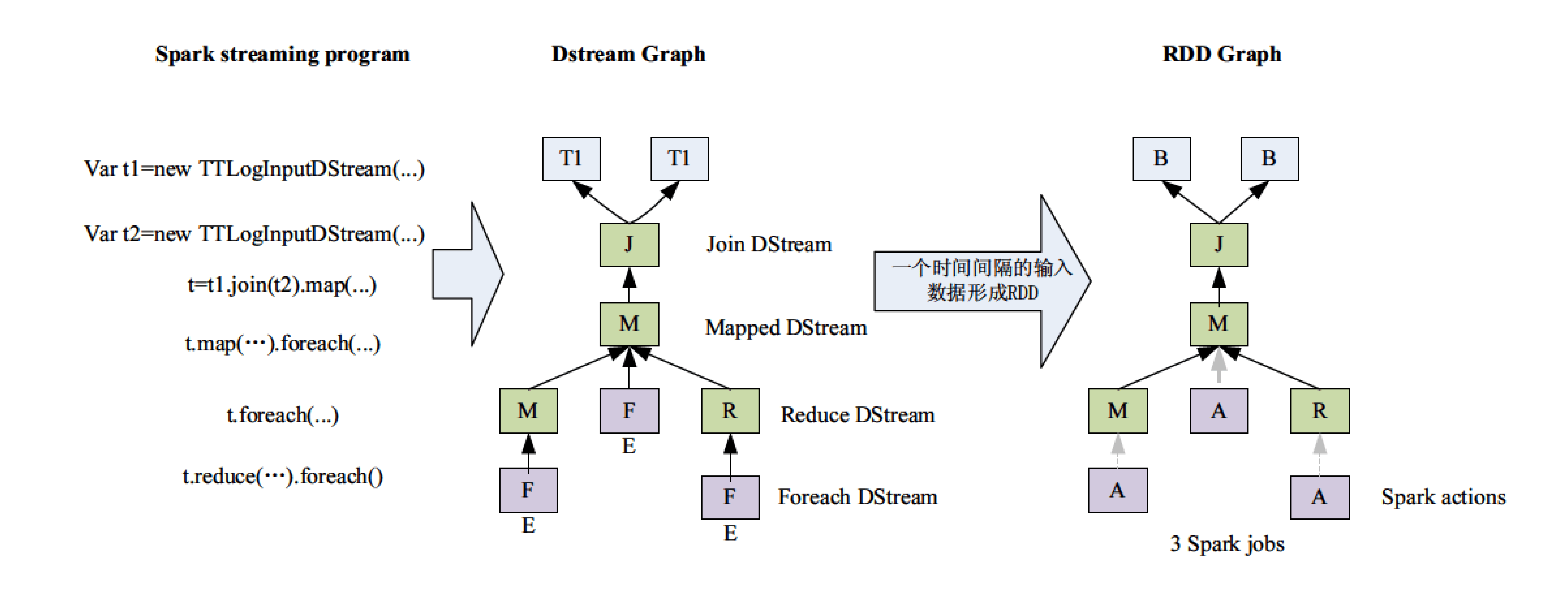


图 4-8　Spark Streaming 程序到 Spark jobs 的转换图

在图 4-8 中，Spark Streaming 把程序中对 DStream 的操作转换为 DStream Graph，图中对于每个时间片，DStream Graph 都会产生一个 RDD Graph；针对每个输出操作（如 print、foreach 等），Spark Streaming 都会创建一个 Spark action；对于每个 Spark action，Spark Streaming 都会产生一个相应的 Spark job，并交给 JobManager。JobManager 中维护着一个 Jobs 队列，Spark job 存储在这个队列中，JobManager 把 Spark job 提交给 Spark Scheduler，Spark Scheduler 负责调度 Task 到相应的 Spark Executor 上执行。

基于云梯 Spark on Yarn 的 Spark Streaming 总体架构如图 4-9 所示。Spark on Yarn 启动后，由 Spark AppMaster 把 Receiver 作为一个 Task 提交给某一个 Spark Executor；Receive 启动后输入数据，生成数据块，然后通知 Spark AppMaster；Spark AppMaster 会根据数据块生成相应的 Job，并把 Job 的 Task 提交给空闲 Spark Executor 执行。图中粗箭头显示被处理的数据流，输入数据流可以是磁盘、网络和 HDFS 等，输出可以是 HDFS、数据库等。

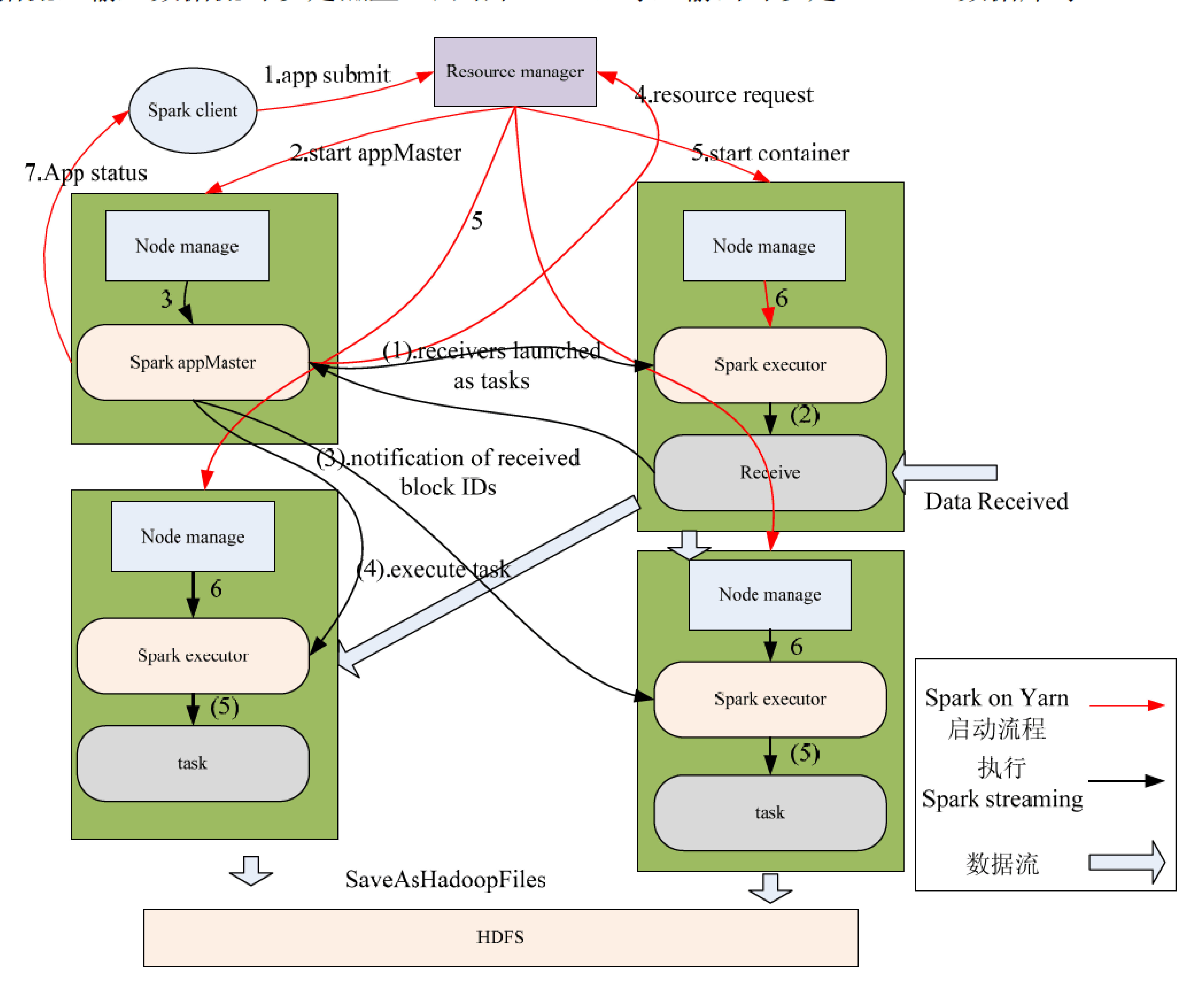


图 4-9　基于云梯 Spark on Yarn 的 Spark Streaming 总体架构

Spark Streaming 的另一大优势在于其容错性，RDD 会记住创建自己的操作，每一批输入数据都会在内存中备份，如果由于某个结点故障导致该结点上的数据丢失，这时可以通过备份的数据在其他结点上重算得到最终的结果。

正如 Spark Streaming 最初的目标一样，它通过丰富的 API 和基于内存的高速计算引擎让用户可以结合流式处理、批处理和交互查询等应用。因此，Spark Streaming 适合一些需要历

史数据和实时数据结合分析的应用场合。当然，对于实时性要求不是特别高的应用也能完全胜任。另外，通过 RDD 的数据重用机制可以得到更高效的容错处理。

4.6 Spark SQL 查询、DataFrames 分布式数据集和 Datasets API

Spark SQL 是 Spark 中处理结构化数据的模块。与基础的 Spark RDD API 不同，Spark SQL 的接口提供了更多关于数据的结构信息和计算任务的运行时信息。在 Spark 内部，Spark SQL 能够用于优化的信息比 RDD API 更多一些。Spark SQL 如今有了三种不同的 API：SQL 语句、DataFrame API 和最新的 Dataset API。不过真正运行计算的时候，无论你使用哪种 API 或语言，Spark SQL 使用的执行引擎都是同一个。这种底层的统一，使开发者可以在不同的 API 之间来回切换，你可以选择一种最自然的方式来表达你的需求[49-51]。

本节中所有的示例都使用 Spark 发布版本中自带的示例数据，并且可以在 spark-shell、pyspark shell 以及 sparkR（R on Spark）shell 中运行。

1. Spark SQL 查询

Spark SQL 的一种用法是执行 SQL 查询。Spark SQL 也可以用于从已安装的 Hive 中读取数据。更多的关于此特性的配置，请参考 Hive Tables[52-54]。当从内部其他编程语言中执行 SQL，Spark SQL 结果将以 Dataset/DataFrame 形式返回。你也可以通过 command-line 或者 JDBC/ODBC 与 SQL 接口进行交互。

2. DataFrames 分布式数据集合

DataFrame 是一种分布式数据集合，每一条数据都由几个命名字段组成。概念上来说，它与关系型数据库的表或者 R 和 Python 中的 data frame 等价，只不过在底层，DataFrame 采用了更多优化。DataFrame 可以从很多数据源（sources）中加载数据并构造得到，如：结构化数据文件、Hive 中的表、外部数据库，或者已有的 RDD。DataFrame API 支持 Scala、Java、Python 和 R。

3. DatasetsAPI

Dataset 是分布式数据集合。Dataset 是 Spark1.6 新增的接口，用以提供 RDDs（强类型，有使用强大的 lambda 函数的能力）和 Spark SQL 经过优化后的执行引擎的优点。Dataset 可以从 JVM 对象进行构造并通过转换函数（如 map、flatmap、filter 等）进行操作。DatasetAPI 支持 Scala 和 Java（http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html）。Python 不支持 Dataset API，但因为 Python 本身的动态性，DatasetAPI 的许多优点都已经可用（比如，你可以通过名字很自然地访问一行的某一个字段，如 row.columnName），R 的情况与此类似。

DataFrame 是 Dataset 组织成命名列的形式。它在概念上相当于关系型数据库中的表，或

者 R/Python 中的数据帧，但是在底层进行了更多的优化。DataFrames 可以从多种数据源创建，例如：结构化数据文件、Hive 中的表、外部数据库或者已存在的 RDDs。DataFrame API 支持 Scala、Java、Python 和 R。在 Scala 和 Java 中 DataFrame 其实是 Dataset 的 RowS 的形式的表示。在 Scala API 中，DataFrame 仅仅是 Dataset[Row]的别名。但在 Java 中，使用者需要使用 Dataset<Row>来表示一个 DataFrame。

4.7 Spark 起始点

4.7.1 SparkSession

在 Spark 中所有功能的切入点是 SparkSession 类。SparkSession 是 Spark2.0 开始提供的，内建了对 Hive 特性的支持，包括使用 HiveQL 写查询语句、调用 Hive UDFs、从 Hive 表读取数据的能力。你不需要事先部署 Hive 就能使用这些特性。

直接使用 SparkSession.builder()就可以创建一个基本的 SparkSession。

（1）Scala

```
Import rog.apache.spark.sql.SparkSession

Val spark = SparkSession
    .builder()
    .appName("Spark SQL Example")
    .config("spark.some.config.option","some-value")
    .getOrCreate()

//For implicit conversions like converting RDDs to DataFrames
Import spark.implicits._
```

（2）Java

```
importorg.apache.spark.sql.SparkSession;

SparkSession spark = SparkSession
 .builder()
  .appName("Java Spark SQL Example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate();
```

（3）Python

```
frompyspark.sqlimport SparkSession

spark = SparkSession\
.builder\
.appName("PythonSQL")\
.config("spark.some.config.option", "some-value")\
.getOrCreate()
```

（4） R

```
sparkR.session(appName = "MyApp", sparkConfig = list(spark.executor.memory =
"1g"))
```

在 Spark 仓库“examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala”中可以找到完整的代码。

4.7.2　SQLContext

Spark SQL 所有的功能入口都是 SQLContext 类及其子类。不过要创建一个 SQLContext 对象，首先需要有一个 SparkContext 对象。

```
val sc: SparkContext// 假设已经有一个 SparkContext 对象
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// 用于包含 RDD 到 DataFrame 隐式转换操作
import sqlContext.implicits._
```

除了 SQLContext 之外，你也可以创建 HiveContext，HiveContext 是 SQLContext 的超集。除了 SQLContext 的功能之外，HiveContext 还支持完整的 HiveQL 语法、使用 UDF，以及对 Hive 表中数据的访问。要使用 HiveContext，你并不需要安装 Hive，而且 SQLContext 能用的数据源，HiveContext 也一样能用。HiveContext 是单独打包的，从而避免了在默认的 Spark 发布版本中包含所有的 Hive 依赖。如果这些依赖对你来说不是问题（不会造成依赖冲突等），建议你在 Spark1.3 之前使用 HiveContext。而后续的 Spark 版本，将会逐渐把 SQLContext 升级到和 HiveContext 功能差不多的状态。

spark.sql.dialect 选项可以指定不同的 SQL 变种（或者叫 SQL 方言）。这个参数可以在 SparkContext.setConf 里指定，也可以通过 SQL 语句的 SET key=value 命令指定。对于 SQLContext，该配置目前唯一的可选值就是“sql”，这个变种使用一个 Spark SQL 自带的简易 SQL 解析器。而对于 HiveContext，spark.sql.dialect 默认值为“hiveql”，当然你也可以将其值设回“sql”。仅就目前而言，HiveSQL 解析器支持更加完整的 SQL 语法，所以大部分情况下，推荐使用 HiveContext。

4.7.3 创建 DataFrame

Spark 应用可以用 SparkContext 创建 DataFrame，所需的数据来源可以是已有的 RDD（existing RDD），或者 Hive 表，或者其他数据源（data sources）。下面的示例从一个 JSON 文件创建一个 DataFrame。

（1）Scala

```
valdf=spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

（2）Java

```
importorg.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
Dataset<Row> df = spark.read().json("examples/src/main/resources/people.json");

// Displays the content of the DataFrame to stdout
df.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

（3）Python

Spark SQL 可以转换 RDD 的行对象到 DataFrame，推断数据类型。行是通过传递键/值对列表参数到行类构造的。这个列表键定义表列的名称和按整个数据库采样推断类型，类似的推论，在 JSON 文件执行。

```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
```

```
# Displays the content of the DataFrame to stdout
df.show()
```

（4） R

```
df <- read.json("examples/src/main/resources/people.json")

# Displays the content of the DataFrame
head(df)

# Another method to print the first few rows and optionally truncate the
printing of long values
showDF(df)
```

4.7.4　无类型的 Dataset 操作（aka DataFrame Operations）

DataFrame 在 Scala、Java、Python 和 R 中为结构化数据操作提供了一个特定领域语言支持。在 Spark2.0 中，在 Scala 和 Java 的 API 中，DataFrame 仅仅是 Dataset 的 RowS 表示。与 Scala/Java 中的强类型的“带类型转换操作”相比，这些操作也可以看作“无类型转换操作”。这里我们提供了一些使用 Dataset 进行结构化数据处理的基本示例。

（1）Scala

```
// This import is needed to use the $-notation
importspark.implicits._
// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1
```

```
df.select($"name",$"age"+1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter($"age">21).show()
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

（2） Java

```
// col("...") is preferable to df.col("...")
importstaticorg.apache.spark.sql.functions.col;

// Print the schema in a tree format
df.printSchema();
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show();
```

```
// +--------+
// |    name|
// +--------+
// |Michael|
// |    Andy|
// | Justin|
// +--------+

// Select everybody, but increment the age by 1
df.select(col("name"),col("age").plus(1)).show();
// +-------+---------+
// |    name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |    Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter(col("age").gt(21)).show();
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show();
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

（3）Python

```
# spark is an existing SparkSession
```

```
# Create the DataFrame
df=spark.read.json("examples/src/main/resources/people.json")

# Show the content of the DataFrame
df.show()
## age  name
## null Michael
## 30   Andy
## 19   Justin

# Print the schema in a tree format
df.printSchema()
## root
## |-- age: long (nullable = true)
## |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
## name
## Michael
## Andy
## Justin

# Select everybody, but increment the age by 1
df.select(df['name'],df['age']+1).show()
## name    (age + 1)
## Michael null
## Andy    31
## Justin  20

# Select people older than 21
df.filter(df['age']>21).show()
## age name
## 30  Andy

# Count people by age
df.groupBy("age").count().show()
## age  count
## null 1
```

```
## 19   1
## 30   1
```

（4） R

```
# Create the DataFrame
df <- read.json("examples/src/main/resources/people.json")

# Show the content of the DataFrame
head(df)
## age  name
## null Michael
## 30   Andy
## 19   Justin

# Print the schema in a tree format
printSchema(df)
## root
## |-- age: long (nullable = true)
## |-- name: string (nullable = true)

# Select only the "name" column
head(select(df,"name"))
## name
## Michael
## Andy
## Justin

# Select everybody, but increment the age by 1
head(select(df, df$name, df$age +1))
## name    (age + 1)
## Michael null
## Andy    31
## Justin  20

# Select people older than 21
head(where(df, df$age >21))
## age name
## 30  Andy
```

```
# Count people by age
head(count(groupBy(df,"age")))
## age  count
## null 1
## 19   1
## 30   1
```

这里我们给出一个结构化数据处理的基本示例：

```
val sc: SparkContext // 已有的 SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// 创建一个 DataFrame
val df = sqlContext.read.json("examples/src/main/resources/people.json")

// 展示 DataFrame 的内容
df.show()
// age  name
// null Michael
// 30   Andy
// 19   Justin

// 打印数据树形结构
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// select "name" 字段
df.select("name").show()
// name
// Michael
// Andy
// Justin

// 展示所有人，但所有人的 age 都加1
df.select(df("name"), df("age") + 1).show()
// name    (age + 1)
// Michael null
// Andy    31
```

```
// Justin  20

// 筛选出年龄大于21的人
df.filter(df("age") > 21).show()
// age name
// 30  Andy

// 计算各个年龄的人数
df.groupBy("age").count().show()
// age  count
// null 1
// 19   1
// 30   1
```

除了简单的字段引用和表达式支持之外，DataFrame 还提供了丰富的工具函数库，包括字符串组装、日期处理、常见的数学函数等。这些函数库的完整列表参见这里：DataFrame Function Reference（http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$）。

4.7.5　编程执行 SQL 查询语句

Sparksession 中的 SQL 函数使得应用可以编程式执行 SQL 查询语句，并且以 DataFrame 形式返回。

1. Scala

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

valsqlDF=spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

2. Java

```
importorg.apache.spark.sql.Dataset;
importorg.apache.spark.sql.Row;
```

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people");

Dataset<Row>sqlDF=spark.sql("SELECT * FROM people");
sqlDF.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

3. Python

```
# spark is an existing SparkSession
df=spark.sql("SELECT * FROM table")
```

4. R

```
df <- sql("SELECT * FROM table")
```

4.7.6 创建 Dataset

Dataset 与 RDD 很像，不同的是它并不使用 Java 序列化或者 Kryo，而是使用特殊的编码器来为网络间的处理或传输的对象进行序列化。对转换一个对象为字节的过程来说，编码器和标准系列化器都是可靠的，编码器的代码是自动生成并且使用了一种特殊数据格式，这种格式允许 Spark 在不需要将字节解码成对象的情况下执行很多操作，如 filtering、sorting 和 hashing 等。

1. Scala

```
// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work
around this limit,
// you can use custom classes that implement the Product interface
caseclassPerson(name:String,age:Long)

// Encoders are created for case classes
valcaseClassDS=Seq(Person("Andy",32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
```

```
// |Andy| 32|
// +----+---+

// Encoders for most common types are automatically provided by importing
spark.implicits._
valprimitiveDS=Seq(1,2,3).toDS()
primitiveDS.map(_+1).collect()// Returns: Array(2, 3, 4)

// DataFrames can be converted to a Dataset by providing a class. Mapping will
be done by name
valpath="examples/src/main/resources/people.json"
valpeopleDS=spark.read.json(path).as[Person]
peopleDS.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

2. Java

```
importjava.util.Arrays;
importjava.util.Collections;
importjava.io.Serializable;

importorg.apache.spark.api.java.function.MapFunction;
importorg.apache.spark.sql.Dataset;
importorg.apache.spark.sql.Row;
importorg.apache.spark.sql.Encoder;
importorg.apache.spark.sql.Encoders;

publicstaticclassPersonimplementsSerializable{
privateStringname;
privateintage;

publicStringgetName(){
returnname;
}
```

```
publicvoidsetName(Stringname){
this.name=name;
}

publicintgetAge(){
returnage;
}

publicvoidsetAge(intage){
this.age=age;
}
}

// Create an instance of a Bean class
Personperson=newPerson();
person.setName("Andy");
person.setAge(32);

// Encoders are created for Java beans
Encoder<Person>personEncoder=Encoders.bean(Person.class);
Dataset<Person>javaBeanDS=spark.createDataset(
Collections.singletonList(person),
personEncoder
);
javaBeanDS.show();
// +---+----+
// |age|name|
// +---+----+
// | 32|Andy|
// +---+----+

// Encoders for most common types are provided in class Encoders
Encoder<Integer>integerEncoder=Encoders.INT();
Dataset<Integer>primitiveDS=spark.createDataset(Arrays.asList(1,2,3),integerEn
coder);
Dataset<Integer>transformedDS=primitiveDS.map(newMapFunction<Integer,Integer>(
){
@Override
```

```
publicIntegercall(Integervalue)throwsException{
returnvalue+1;
}
},integerEncoder);
transformedDS.collect();// Returns [2, 3, 4]

// DataFrames can be converted to a Dataset by providing a class. Mapping
based on name
Stringpath="examples/src/main/resources/people.json";
Dataset<Person>peopleDS=spark.read().json(path).as(personEncoder);
peopleDS.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

举例：

```
// 对普通类型数据的 Encoder 是由 importing sqlContext.implicits._ 自动提供的
val ds = Seq(1, 2, 3).toDS()
ds.map(_ + 1).collect() // 返回: Array(2, 3, 4)

// 以下这行不仅定义了 case class，同时也自动为其创建了 Encoder
case class Person(name: String, age: Long)
val ds = Seq(Person("Andy", 32)).toDS()

// DataFrame 只需提供一个和数据 schema 对应的 class 即可转换为 Dataset。Spark 会根据字段名
进行映射。
val path = "examples/src/main/resources/people.json"
val people = sqlContext.read.json(path).as[Person]
```

4.7.7　和 RDD 互操作

Spark SQL 支持两种将已存在的 RDD 转化为 Dataset 的方法。第一种使用反射机制，它包含指定类型对象 RDD 的 schema。这种基于反射机制的方法使代码更简洁，而且如果你事先知道数据 schema，推荐使用这种方式；第二种创建 Dataset 的方法是通过编程接口建立一个结构，然后将它应用于一个存在的 RDD。虽然这种方法更加烦琐，但它允许你在运行之前不知道其中的列和对应的类型的情况下构建 Dataset。

1. 利用反射推导 schema

Spark SQL 的 Scala 接口支持自动地将一个包含 case class 的 RDD 转换为 DataFrame。这个 case class 定义了表结构。Case class 的参数名是通过反射机制读取，然后变成列名。Case class 可以嵌套或者包含像 Seq 或 Array 之类的复杂类型。这个 RDD 可以隐式地转换为一个 DataFrame，然后被注册为一张表。这个表可以随后被 SQL 的 statement 使用。

（1）Scala

```
importorg.apache.spark.sql.catalyst.encoders.ExpressionEncoder
importorg.apache.spark.sql.Encoder

// For implicit conversions from RDDs to DataFrames
importspark.implicits._

// Create an RDD of Person objects from a text file, convert it to a Dataframe
valpeopleDF=spark.sparkContext
.textFile("examples/src/main/resources/people.txt")
.map(_.split(","))
.map(attributes=>Person(attributes(0),attributes(1).trim.toInt))
.toDF()
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql methods provided by Spark
valteenagersDF=spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13
AND 19")

// The columns of a row in the result can be accessed by field index
teenagersDF.map(teenager=>"Name: "+teenager(0)).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+

// or by field name
teenagersDF.map(teenager=>"Name: "+teenager.getAs[String]("name")).show()
// +------------+
// |       value|
// +------------+
```

```
// |Name: Justin|
// +------------+

// No pre-defined encoders for Dataset[Map[K,V]], define explicitly
implicitvalmapEncoder=org.apache.spark.sql.Encoders.kryo[Map[String, Any]]
// Primitive types and case classes can be also defined as
implicitvalstringIntMapEncoder:Encoder[Map[String, Int]]=ExpressionEncoder()

// row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]
teenagersDF.map(teenager=>teenager.getValuesMap[Any](List("name","age"))).coll
ect()
// Array(Map("name" ->"Justin", "age" -> 19))
```

（2）Java

Spark SQL 支持自动转换成一个 DataFrame 的 JavaBeans RDD。BeanInfo 利用反射定义表的架构。尽管 Spark SQL 支持嵌套的 JavaBeans、列表或数组字段，但目前 Spark SQL 不支持包含域映射 JavaBeans。你可以创建一个具有其领域的 getter 和 setter，并实现序列化接口类的 JavaBean。

```
importorg.apache.spark.api.java.JavaRDD;
importorg.apache.spark.api.java.function.Function;
importorg.apache.spark.api.java.function.MapFunction;
importorg.apache.spark.sql.Dataset;
importorg.apache.spark.sql.Row;
importorg.apache.spark.sql.Encoder;
importorg.apache.spark.sql.Encoders;

// Create an RDD of Person objects from a text file
JavaRDD<Person>peopleRDD=spark.read()
.textFile("examples/src/main/resources/people.txt")
.javaRDD()
.map(newFunction<String,Person>(){
@Override
publicPersoncall(Stringline)throwsException{
String[]parts=line.split(",");
Personperson=newPerson();
person.setName(parts[0]);
person.setAge(Integer.parseInt(parts[1].trim()));
returnperson;
```

```
}
});

// Apply a schema to an RDD of JavaBeans to get a DataFrame
Dataset<Row>peopleDF=spark.createDataFrame(peopleRDD,Person.class);
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people");

// SQL statements can be run by using the sql methods provided by spark
Dataset<Row>teenagersDF=spark.sql("SELECT name FROM people WHERE age BETWEEN
13 AND 19");

// The columns of a row in the result can be accessed by field index
Encoder<String>stringEncoder=Encoders.STRING();
Dataset<String>teenagerNamesByIndexDF=teenagersDF.map(newMapFunction<Row,Strin
g>(){
@Override
publicStringcall(Rowrow)throwsException{
return"Name: "+row.getString(0);
}
},stringEncoder);
teenagerNamesByIndexDF.show();
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+

// or by field name
Dataset<String>teenagerNamesByFieldDF=teenagersDF.map(newMapFunction<Row,Strin
g>(){
@Override
publicStringcall(Rowrow)throwsException{
return"Name: "+row.<String>getAs("name");
}
},stringEncoder);
teenagerNamesByFieldDF.show();
// +------------+
// |       value|
```

```
// +------------+
// |Name: Justin|
// +------------+
```

（3）Python

Spark SQL 可以转换 RDD 的行对象到 DataFrame，推断数据类型。行是通过传递键/值对列表参数到行类构造的。这个列表键定义表列的名称和按整个数据库采样推断类型，类似的推论，在 JSON 文件执行。

```
# spark is an existing SparkSession.
frompyspark.sqlimportRow
sc=spark.sparkContext

# Load a text file and convert each line to a Row.
lines=sc.textFile("examples/src/main/resources/people.txt")
parts=lines.map(lambdal:l.split(","))
people=parts.map(lambdap:Row(name=p[0],age=int(p[1])))

# Infer the schema, and register the DataFrame as a table.
schemaPeople=spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table.
teenagers=spark.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# The results of SQL queries are RDDs and support all the normal RDD
operations.
teenNames=teenagers.map(lambdap:"Name: "+p.name)
forteenNameinteenNames.collect():
print(teenName)
```

举例：

```
// sc 是已有的 SparkContext 对象
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// 为了支持 RDD 到 DataFrame 的隐式转换
import sqlContext.implicits._

// 定义一个 case class.
// 注意：Scala 2.10的 case class 最多支持22个字段，要绕过这一限制，
// 你可以使用自定义 class，并实现 Product 接口。当然，你也可以改用编程方式定义 schema
```

```
case class Person(name: String, age: Int)

// 创建一个包含 Person 对象的 RDD，并将其注册成 table
val people =
sc.textFile("examples/src/main/resources/people.txt").map(_.split(",")).map(p
=> Person(p(0), p(1).trim.toInt)).toDF()
people.registerTempTable("people")

// sqlContext.sql 方法可以直接执行 SQL 语句
val teenagers = sqlContext.sql("SELECT name, age FROM people WHERE age >= 13
AND age <= 19")

// SQL 查询的返回结果是一个 DataFrame，且能够支持所有常见的 RDD 算子
// 查询结果中每行的字段可以按字段索引访问：
teenagers.map(t =>"Name: " + t(0)).collect().foreach(println)

// 或者按字段名访问：
teenagers.map(t =>"Name: " +
t.getAs[String]("name")).collect().foreach(println)

// row.getValuesMap[T] 会一次性返回多列，并以 Map[String, T]为返回结果类型
teenagers.map(_.getValuesMap[Any](List("name",
"age"))).collect().foreach(println)
// 返回结果：Map("name" ->"Justin", "age" -> 19)
```

2. 编程方式定义 Schema

当 case class 不能被事先定义（比如记录的结构被编码为字符串，或者对不同的用户，文本数据集被不同的解析并进行字段投影），DataFrame 可以通过以下 3 个方法实现编程创建：

- 从原始 RDD 创建 RowS 形式的 RDD。
- 以 StructType 创建匹配上一个方法中 RowS 形式的 RDD 的模式。
- 通过 SparkSession 提供的 createDataFrame 方法将模式应用于 RowS 形式的 RDD。

（1）Scala

```
importorg.apache.spark.sql.types._

// Create an RDD
valpeopleRDD=spark.sparkContext.textFile("examples/src/main/resources/people.t
xt")
```

```
// The schema is encoded in a string
valschemaString="name age"

// Generate the schema based on the string of schema
valfields=schemaString.split("")
.map(fieldName=>StructField(fieldName,StringType,nullable=true))
valschema=StructType(fields)

// Convert records of the RDD (people) to Rows
valrowRDD=peopleRDD
.map(_.split(","))
.map(attributes=>Row(attributes(0),attributes(1).trim))

// Apply the schema to the RDD
valpeopleDF=spark.createDataFrame(rowRDD,schema)

// Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")

// SQL can be run over a temporary view created using DataFrames
valresults=spark.sql("SELECT name FROM people")

// The results of SQL queries are DataFrames and support all the normal RDD
operations
// The columns of a row in the result can be accessed by field index or by
field name
results.map(attributes=>"Name: "+attributes(0)).show()
// +-------------+
// |        value|
// +-------------+
// |Name: Michael|
// |   Name: Andy|
// | Name: Justin|
// +-------------+
```

（2）Java

当 JavaBean 类不能提前被定义时，DataFrame 可以如上面 Scala 类似处理，通过 3 个步骤实现编程创建。

```
importjava.util.ArrayList;
importjava.util.List;

importorg.apache.spark.api.java.JavaRDD;
importorg.apache.spark.api.java.function.Function;

importorg.apache.spark.sql.Dataset;
importorg.apache.spark.sql.Row;

importorg.apache.spark.sql.types.DataTypes;
importorg.apache.spark.sql.types.StructField;
importorg.apache.spark.sql.types.StructType;

// Create an RDD
JavaRDD<String>peopleRDD=spark.sparkContext()
.textFile("examples/src/main/resources/people.txt",1)
.toJavaRDD();

// The schema is encoded in a string
StringschemaString="name age";

// Generate the schema based on the string of schema
List<StructField>fields=newArrayList<>();
for(StringfieldName:schemaString.split("")){
StructFieldfield=DataTypes.createStructField(fieldName,DataTypes.StringType,tr
ue);
fields.add(field);
}
StructTypeschema=DataTypes.createStructType(fields);

// Convert records of the RDD (people) to Rows
JavaRDD<Row>rowRDD=peopleRDD.map(newFunction<String,Row>(){
@Override
publicRowcall(Stringrecord)throwsException{
String[]attributes=record.split(",");
returnRowFactory.create(attributes[0],attributes[1].trim());
}
});
```

```
// Apply the schema to the RDD
Dataset<Row>peopleDataFrame=spark.createDataFrame(rowRDD,schema);

// Creates a temporary view using the DataFrame
peopleDataFrame.createOrReplaceTempView("people");

// SQL can be run over a temporary view created using DataFrames
Dataset<Row>results=spark.sql("SELECT name FROM people");

// The results of SQL queries are DataFrames and support all the normal RDD
operations
// The columns of a row in the result can be accessed by field index or by
field name
Dataset<String>namesDS=results.map(newMapFunction<Row,String>(){
@Override
publicStringcall(Rowrow)throwsException{
return"Name: "+row.getString(0);
}
},Encoders.STRING());
namesDS.show();
// +-------------+
// |        value|
// +-------------+
// |Name: Michael|
// |   Name: Andy|
// | Name: Justin|
// +-------------+
```

（3）Python

若关键字参数字典不能被提前定义，则 DataFrame 可以如 Scala 和 Java 类似处理，通过 3 个步骤实现编程创建。

```
# Import SparkSession and data types
frompyspark.sql.typesimport*

# spark is an existing SparkSession.
sc=spark.sparkContext

# Load a text file and convert each line to a tuple.
```

```
lines=sc.textFile("examples/src/main/resources/people.txt")
parts=lines.map(lambdal:l.split(","))
people=parts.map(lambdap:(p[0],p[1].strip()))

# The schema is encoded in a string.
schemaString="name age"

fields=[StructField(field_name,StringType(),True)forfield_nameinschemaString.s
plit()]
schema=StructType(fields)

# Apply the schema to the RDD.
schemaPeople=spark.createDataFrame(people,schema)

# Creates a temporary view using the DataFrame
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table.
results=spark.sql("SELECT name FROM people")

# The results of SQL queries are RDDs and support all the normal RDD
operations.
names=results.map(lambdap:"Name: "+p.name)
fornameinnames.collect():
print(name)
```

举例：

```
// sc 是已有的 SparkContext 对象
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// 创建一个 RDD
val people = sc.textFile("examples/src/main/resources/people.txt")

// 数据的 schema 被编码与一个字符串中
val schemaString = "name age"

// Import Row.
import org.apache.spark.sql.Row;

// Import Spark SQL 各个数据类型
```

```
import org.apache.spark.sql.types.{StructType,StructField,StringType};

// 基于前面的字符串生成 schema
val schema = StructType(
   schemaString.split("").map(fieldName => StructField(fieldName, StringType,
true)))

// 将 RDD[people]的各个记录转换为 Rows，即：得到一个包含 Row 对象的 RDD
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))

// 将 schema 应用到包含 Row 对象的 RDD 上，得到一个 DataFrame
val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema)

// 将 DataFrame 注册为 table
peopleDataFrame.registerTempTable("people")

// 执行 SQL 语句
val results = sqlContext.sql("SELECT name FROM people")

// SQL 查询的结果是 DataFrame，且能够支持所有常见的 RDD 算子
// 并且其字段可以以索引访问，也可以用字段名访问
results.map(t =>"Name: " + t(0)).collect().foreach(println)
```

4.8 Spark 数据源

Spark SQL 支持基于 DataFrame 操作一系列不同的数据源。DataFrame 既可以当成一个普通 RDD 来操作，也可以将其注册成一个临时表来查询。把 DataFrame 注册为表之后，你就可以基于这个表执行 SQL 语句了。本节将描述加载和保存数据的一些通用方法，包含了不同的 Spark 数据源，然后深入介绍一下内建数据源可用选项。

4.8.1 通用加载/保存函数

在最简单的情况下，所有操作都会以默认类型数据源来加载数据（默认是 Parquet，除非修改了 spark.sql.sources.default 配置）。

```
val df = sqlContext.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

1. 手动指定选项

你也可以手动指定数据源，并设置一些额外的选项参数。数据源可由其全名指定（如，org.apache.spark.sql.parquet），而对于内建支持的数据源，可以使用简写名（json、parquet、

jdbc）。任意类型数据源创建的 DataFrame 都可以用下面这种语法转成其他类型的数据格式。

（1）Scala

```
val peopleDF =
spark.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.select("name",
"age").write.format("parquet").save("namesAndAges.parquet")
```

（2）Java

```
Dataset<Row> peopleDF =
spark.read().format("json").load("examples/src/main/resources/people.json");
peopleDF.select("name",
"age").write().format("parquet").save("namesAndAges.parquet");
```

（3）Python

```
df = spark.read.load("examples/src/main/resources/people.json", format="json")
df.select("name", "age").write.save("namesAndAges.parquet", format="parquet")
```

（4）R

```
df <- read.df("examples/src/main/resources/people.json", "json")
namesAndAges <- select(df, "name", "age")
write.df(namesAndAges, "namesAndAges.parquet", "parquet")
```

2. 在文件上直接执行 SQL

除了使用读取 API，加载一个文件到 SATAFrame，然后查询它的方式，你同样可以通过 SQL 直接查询文件。

（1）Scala

```
valsqlDF = spark.sql("SELECT * FROM
parquet.`examples/src/main/resources/users.parquet`")
```

（2）Java

```
Dataset<Row> sqlDF =
  spark.sql("SELECT * FROM
parquet.`examples/src/main/resources/users.parquet`");
```

（3）Python

```
df = spark.sql("SELECT * FROM
parquet.`examples/src/main/resources/users.parquet`")
```

（4）R

```
df <- sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`")
```

3. 保存模式

保存操作可选 SaveMode，它指定了如何处理现有的数据。需要重视的一点是这些保存模式没有使用任何锁，并且不具有原子性。此外，当执行 Overwrite 时，数据将先被删除，然后写出新数据。保存模式如表 4-8 所示。

表 4-8 保存模式

Scala/Java	其他语言	含义
SaveMode.ErrorIfEcists（默认）	error（默认）	保存 DataFrame 到数据源时，如果数据已经存在，将抛出一个异常
SaveMode.Append	append	保存 DataFrame 到数据源时，如果数据/表存在时，DataFrame 的内容将追加到已存在的数据后
SaveMode.Overwrite	overwrite	Overwrite 模式意味着当保存一个 DataFrame 到数据源时，如果数据/表已经存在，存在的数据将会被 DataFrame 的内容覆盖
SaveMode.Ignore	ignore	Ignore 模式意味着当保存一个 DataFrame 到数据源时，如果数据已经存在，保存操作将不会保存 DataFrame 的内容，并且不会改变原数据。这与 SQL 中的 CREATE TABLE IF NOT EXISTS 相似

4. 保存到持久化表

也可以通过 saveAsTable 命令将 DataFrame 作为持久化表保存到 Hive 元数据库中。注意使用此特性时不需要事先部署 Hive。Spark 将为你创建一个默认的本地 Hive 元数据库（使用 Derby）。不同于 createOrReplaceTempView 命令，saveAsTable 将具体化 DataFrame 的内容并且在 Hive 元数据库中创建一个指向数据的指针。在你保持你的连接到相同的元数据库时，当你的 Spark 程序重启后持久化表依然会存在。通过在 SparkSession 上使用表名调用 table 命令，可以创建用于持久化表的 DataFrame。

默认的 saveAsTable 将会创建一个“托管表”，意味着数据的位置将由元数据库控制。托管表也有他们自己的数据，当对应的表被删除时这些数据会一并删除。

4.8.2 Parquet 文件

Parquet 是一种流行的列式存储格式。Spark SQL 提供对 Parquet 文件的读写支持，而且 Parquet 文件能够自动保存原始数据的 schema。写 Parquet 文件的时候，所有的字段都会自动转成 nullable，以便向后兼容。

1. 编程方式加载数据

仍然使用上面例子中的数据：

```
// 我们继续沿用之前例子中的 sqlContext 对象
// 为了支持 RDD 隐式转成 DataFrame
import sqlContext.implicits._

val people: RDD[Person] = ... // 和上面例子中相同，一个包含 case class 对象的 RDD

// 该 RDD 将隐式转成 DataFrame，然后保存为 parquet 文件
people.write.parquet("people.parquet")

// 读取上面保存的 Parquet 文件(多个文件 - Parquet 保存完其实是很多个文件)。Parquet 文件是自
描述的，文件中保存了 schema 信息
// 加载 Parquet 文件，并返回 DataFrame 结果
val parquetFile = sqlContext.read.parquet("people.parquet")

// Parquet 文件（多个）可以注册为临时表，然后在 SQL 语句中直接查询
parquetFile.registerTempTable("parquetFile")
val teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13
AND age <= 19")
teenagers.map(t =>"Name: " + t(0)).collect().foreach(println)
```

（1）Scala

```
// Encoders for most common types are automatically provided by importing
spark.implicits._
import spark.implicits._

val peopleDF = spark.read.json("examples/src/main/resources/people.json")

// DataFrames can be saved as Parquet files, maintaining the schema
information
peopleDF.write.parquet("people.parquet")

// Read in the parquet file created above
// Parquet files are self-describing so the schema is preserved
// The result of loading a Parquet file is also a DataFrame
val parquetFileDF = spark.read.parquet("people.parquet")

// Parquet files can also be used to create a temporary view and then used in
SQL statements
parquetFileDF.createOrReplaceTempView("parquetFile")
```

```
val namesDF = spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13 AND
19")
namesDF.map(attributes =>"Name: " + attributes(0)).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

（2）Java

```
importorg.apache.spark.api.java.function.MapFunction;
importorg.apache.spark.sql.Encoders;
importorg.apache.spark.sql.Dataset;
importorg.apache.spark.sql.Row;

Dataset<Row>peopleDF=spark.read().json("examples/src/main/resources/people.jso
n");

// DataFrames can be saved as Parquet files, maintaining the schema
information
peopleDF.write().parquet("people.parquet");

// Read in the Parquet file created above.
// Parquet files are self-describing so the schema is preserved
// The result of loading a parquet file is also a DataFrame
Dataset<Row>parquetFileDF=spark.read().parquet("people.parquet");

// Parquet files can also be used to create a temporary view and then used in
SQL statements
parquetFileDF.createOrReplaceTempView("parquetFile");
Dataset<Row>namesDF=spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN
13 AND 19");
Dataset<String>namesDS=namesDF.map(newMapFunction<Row,String>(){
publicStringcall(Rowrow){
return"Name: "+row.getString(0);
}
},Encoders.STRING());
namesDS.show();
// +------------+
```

```
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

（3）Python

```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

# DataFrames can be saved as Parquet files, maintaining the schema information.
peopleDF.write.parquet("people.parquet")

# Read in the Parquet file created above.
# Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a DataFrame.
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in
SQL statements.
parquetFile.createOrReplaceTempView("parquetFile")
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <=
19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

（4）R

```
df <- read.df("examples/src/main/resources/people.json","json")

# SparkDataFrame can be saved as Parquet files, maintaining the schema
information.
write.parquet(df,"people.parquet")

# Read in the Parquet file created above. Parquet files are self-describing so
the schema is preserved.
# The result of loading a parquet file is also a DataFrame.
parquetFile <- read.parquet("people.parquet")
```

```
# Parquet files can also be used to create a temporary view and then used in
SQL statements.
createOrReplaceTempView(parquetFile,"parquetFile")
teenagers <- sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
head(teenagers)
##     name
## 1 Justin

# We can also run custom R-UDFs on Spark DataFrames. Here we prefix all the
names with "Name:"
schema <- structType(structField("name","string"))
teenNames <- dapply(df,function(p){cbind(paste("Name:", p$name))}, schema)
for(teenName in collect(teenNames)$name){
cat(teenName,"\n")
}
## Name: Michael
## Name: Andy
## Name: Justin
```

（5）SQL

```
CREATETEMPORARYVIEWparquetTable
USINGorg.apache.spark.sql.parquet
OPTIONS(
path"examples/src/main/resources/people.parquet"
)
SELECT*FROMparquetTable
```

2. 分区发现

表分区是 Hive 等系统中常用的优化方法。在一个分区表中，数据常常存放在不同的目录中，根据分区列的值的不同，编码了每个分区目录不同的路径。目前 parquet 数据源已经可以自动地发现和推断分区信息。例如，我们可以用下面的目录结构存储所有我们以前经常使用的数据到分区表，只需要额外地添加两个列 gender 和 country 作为分区列：

```
path
└── to
    └── table
        ├── gender=male
        │   ├── ...
```

```
        |    |
        |    ├─── country=US
        |    |    └─── data.parquet
        |    ├─── country=CN
        |    |    └─── data.parquet
        |    └─── ...
        └─── gender=female
            ├─── ...
            |
            ├─── country=US
            |    └─── data.parquet
            ├─── country=CN
            |    └─── data.parquet
            └─── ...
```

在这个例子中，如果需要读取 Parquet 文件数据，我们只需要把 path/to/table 作为参数传给 SQLContext.read.parquet 或者 SQLContext.read.load。Spark SQL 能够自动地从路径中提取出分区信息，随后返回的 DataFrame 的 schema 如下：

```
root
|-- name: string (nullable = true)
|-- age: long (nullable = true)
|-- gender: string (nullable = true)
|-- country: string (nullable = true)
```

注意分区列的数据类型是自动推断的。目前支持数值型数据和字符串型数据。有时候用户并不想自动推断分区列的数据类型，这种情况下，可以通过配置 spark.sql.sources.partitionColumnTypeInference.enabled 参数来配置自动类型推断，默认情况下是 true。当关闭类型推断后，分区列的类型将为字符串型。

从 Spark1.6.0 开始，在默认情况下，只在给定的路径下进行分区发现。在上述的例子中，如果用户将 path/to/table/gender=male 传给 SparkSession.read.parquet 或者 SparkSession.read.load，gender 将会被认为是分区列。如果用户需要指定分区开始的基础路径，可以将 basePath 设置到数据源选项。例如，当 path/to/table/gender=male 是数据的路径，并且用户设置 basePath 为 path/to/table，gender 将作为分区列。

3. 模式（Schema）合并

与 ProtocolBuffer、Avro 和 Thrift 类似，Parquet 同样支持 schema 的演变。用户可以以一个简单点的 schema 开始，然后在需要时逐渐地添加更多列到 schema。使用这种方法，用户将最终得到由不同的但是相互兼容的 schema 构成的多个 Parquet 文件。Parquet 数据源目前可以自动地检测这种情况并且合并这些文件的 schema。

由于合并 schema 是相对代价较大的操作，而且在大多数情况下并不需要这样，从 1.5.0 开始我们默认将它关闭，你可以通过以下方法使它生效：

- 在读取 Parquet 文件时（就像下面的例子）设置数据源操作 mergeSchema 为 true。
- 设置全局 SQL 选项 spark.sql.parquet.mergeSchema 为 true。

```
// 继续沿用之前的 sqlContext 对象
// 为了支持 RDD 隐式转换为 DataFrame
import sqlContext.implicits._

// 创建一个简单的 DataFrame，存到一个分区目录中
val df1 = sc.makeRDD(1 to 5).map(i => (i, i * 2)).toDF("single", "double")
df1.write.parquet("data/test_table/key=1")

// 创建另一个 DataFrame 放到新的分区目录中，
// 并增加一个新字段，丢弃一个老字段
val df2 = sc.makeRDD(6 to 10).map(i => (i, i * 3)).toDF("single", "triple")
df2.write.parquet("data/test_table/key=2")

// 读取分区表
val df3 = sqlContext.read.option("mergeSchema",
"true").parquet("data/test_table")
df3.printSchema()

// 最终的 schema 将由3个字段组成（single，double，triple）
// 并且分区键出现在目录路径中
// root
// |-- single: int (nullable = true)
// |-- double: int (nullable = true)
// |-- triple: int (nullable = true)
// |-- key : int (nullable = true)
```

4. Hive metastore Parquet table 转换

在读写 Hive metastore Parquet 表时，Spark SQL 用的是内部的 Parquet 支持库，而不是 Hive SerDe，因为这样性能更好。这一行为是由 spark.sql.hive.convertMetastoreParquet 配置项来控制的，而且默认是启用的。

（1）Hive/Parquet schema 调和

从表的 schema 处理的角度来看，Hive 和 Parquet 有两点关键的不同之处。

- Hive 是类型敏感的，而 Parquet 并不是。
- Hive 中所有列都是非空的，而 Parquet 中非空是很重要的特性。

因为这个原因，当我们需要将 Hive 元存储转换为 Spark SQL Parquet 表中的 Parquet 表时，我们需要调节 Hive 元存储的 schema 和 Parquet 的 schema。调节规则如下：

- 不管是否可为空值，两种 schema 中具有相同名字的字段必须具有相同的数据类型。这种调节字段应该有与 Parquet 一方相同的数据类型，因此可为空值的特性很重要。
- 调节的 schema 准确地包含在 Hive 元存储 schema 中定义的字段。
- 任何只在 Parquet schema 中出现的字段都会在调节 schema 中被丢弃。
- 任何只出现在 Hive 元存储 schema 中的字段都会在调节 schema 中被添加为可为空的字段。

（2）刷新元数据

为了更好的性能，Spark SQL 会缓存 Parquet 元数据。当 Hive 元存储 Parquet 表转换操作可用时，这些被转换的表的元数据同样被缓存。如果这些表被 Hive 或者外部工具更新，你需要手动更新元数据以保持其一致性。

```
// 注意，这里 sqlContext 是一个 HiveContext
sqlContext.refreshTable("my_table")
Scala:
// spark is an existing SparkSession
spark.catalog.refreshTable("my_table")
Java:
// spark is an existing SparkSession
spark.catalog().refreshTable("my_table");
Python:
# spark is an existing SparkSession
spark.catalog.refreshTable("my_table")
Sql:
REFRESH TABLE my_table;
```

5. 配置

Parquet 的配置可以使用 SparkSession 中的 setConf 方法进行，或者使用 SQL 执行 SET key=value 命令。Parquet 配置如表 4-9 所示。

表 4-9　Parquet 配置

属性名	默认值	含义
spark.sql.parquet.binaryAsString	false	有些老系统，如：特定版本的 Impala、Hive，或者老版本的 Spark SQL，不区分二进制数据和字符串类型数据。这个标志的意思是，让 Spark SQL 把二进制数据当字符串处理，以兼容老系统
spark.sql.parquet.int96AsTimestamp	true	有些老系统，如：特定版本的 Impala、Hive，把时间戳存成 INT96。这个配置的作用是，让 Spark SQL 把这些 INT96 解释为 timestamp，以兼容老系统
spark.sql.parquet.cacheMetadata	true	缓存 Parquet schema 元数据。可以提升查询静态数据的速度

（续表）

属性名	默认值	含义
spark.sql.parquet.compression.codec	gzip	设置 Parquet 文件的压缩编码格式。可接受的值有：uncompressed、snappy、gzip（默认），lzo
spark.sql.parquet.filterPushdown	true	启用过滤器下推优化，可以将过滤条件尽量推到最下层，已取得性能提升
spark.sql.hive.convertMetastoreParquet	true	如果禁用，Spark SQL 将使用 Hive SerDe，而不是内建的对 Parquet tables 的支持
spark.sql.parquet.output.committer.class	org.apache.parquet.hadoop.ParquetOutputCommitter	Parquet 使用的数据输出类。这个类必须是 org.apache.hadoop.mapreduce.OutputCommitter 的子类。一般来说，它也应该是 org.apache.parquet.hadoop.ParquetOutputCommitter 的子类。 注意： （1）如果启用 spark.speculation，这个选项将被自动忽略。 （2）这个选项必须用 hadoop configuration 设置，而不是 Spark SQLConf。 （3）这个选项会覆盖 spark.sql.sources.outputCommitterClass。 Spark SQL 有一个内建的 org.apache.spark.sql.parquet. DirectParquetOutputCommitter，这个类在输出到 S3 的时候比默认的 ParquetOutputCommitter 类效率高
spark.sql.parquet.mergeSchema	false	如果设为 true，那么 Parquet 数据源将会 merge 所有数据文件的 schema；否则，schema 是从 summary file 获取的（如果 summary file 没有设置，则随机选一个）

4.8.3　JSON 数据集

Spark SQL 可以自动推断 JSON 数据集的 schema 并且加载为 Dataset[Row]，可以对 String 类型的 RDD 或者 JSON 文件使用 SparkSession.read.json()来实现这种转换。注意，这里的 JSON 文件不是通常意义的 JSON 文件，每一行必须包含分离的、完整有效的 JSON 对象。因此，不支持常用的多行式 JSON 文件。

```
// sc 是已有的 SparkContext 对象
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// 数据集是由路径指定的
// 路径既可以是单个文件，也可以还是存储文本文件的目录
val path = "examples/src/main/resources/people.json"
val people = sqlContext.read.json(path)

// 推导出来的 schema，可由 printSchema 打印出来
people.printSchema()
// root
//  |-- age: integer (nullable = true)
//  |-- name: string (nullable = true)

// 将 DataFrame 注册为 table
```

```
people.registerTempTable("people")

// 运行 SQL 语句
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND
age <= 19")

// 另一种方法是，用一个包含 JSON 字符串的 RDD 来创建 DataFrame
val anotherPeopleRDD = sc.parallelize(
"""{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""":: Nil)
val anotherPeople = sqlContext.read.json(anotherPeopleRDD)
```

注意，RDD[String]中每一个元素必须是一个字符串形式的 JSON 对象。可以在 Spark 仓库的 “examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala”找到完整的代码。

4.8.4 Hive 表

Spark SQL 同样支持从 Apache Hive 中读写数据。但是，自从 Hive 有大量依赖之后，这些依赖就不包括在 Spark 发布版中了。如果 Hive 的依赖可以在环境变量中找到，Spark 将自动加载它们。注意这些 Hive 依赖项同样必须在每个 Worker 节点上存在，因为他们需要访问 Hive 序列化和反序列化库以便可以访问 Hive 中存储的数据。可以在 conf/目录中的 hive-site.xml、core-site.xml（安全配置）和 hdfs-site.xml（HDFS 配置）这几个文件中进行配置。

当在 Hive 上工作时，必须实例化 SparkSession 对 Hive 的支持，包括对持久化 Hive 元存储的连通性、对 Hive 序列化反序列化、Hive 用户自定义函数的支持。当没有在 hive-site.xml 配置时，context 会自动在当前目录创建 metastore_db 并且创建一个被 spark.sql.warehouse.dir 配置的目录，默认在 Spark 应用启动的当前目录的 spark-warehouse 中配置。注意从 Spark2.0.0 开始，hive-site.xml 中的 hive.metastore.warehouse.dir 参数被弃用。作为替代，使用 spark.sql.warehouse.dir 来指定仓库中数据库的位置。你可能需要授予写权限给启动 Spark 应用的用户。

1. Scala

```
importorg.apache.spark.sql.Row
importorg.apache.spark.sql.SparkSession

caseclassRecord(key:Int,value:String)

// warehouseLocation points to the default location for managed databases and
tables
valwarehouseLocation="file:${system:user.dir}/spark-warehouse"

valspark=SparkSession
```

```
.builder()
.appName("Spark Hive Example")
.config("spark.sql.warehouse.dir",warehouseLocation)
.enableHiveSupport()
.getOrCreate()

importspark.implicits._
importspark.sql

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE
src")

// Queries are expressed in HiveQL
sql("SELECT * FROM src").show()
// +---+-------+
// |key|  value|
// +---+-------+
// |238|val_238|
// | 86| val_86|
// |311|val_311|
// ...

// Aggregation queries are also supported.
sql("SELECT COUNT(*) FROM src").show()
// +--------+
// |count(1)|
// +--------+
// |    500 |
// +--------+

// The results of SQL queries are themselves DataFrames and support all normal
functions.
valsqlDF=sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

// The items in DaraFrames are of type Row, which allows you to access each
column by ordinal.
valstringsDS=sqlDF.map{
caseRow(key:Int,value:String)=>s"Key: $key, Value: $value"
```

```
}
stringsDS.show()
// +--------------------+
// |               value|
// +--------------------+
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// ...

// You can also use DataFrames to create temporary views within a SparkSession.
valrecordsDF=spark.createDataFrame((1to100).map(i=>Record(i,s"val_$i")))
recordsDF.createOrReplaceTempView("records")

// Queries can then join DataFrame data with data stored in Hive.
sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show()
// +---+------+---+------+
// |key| value|key| value|
// +---+------+---+------+
// |  2| val_2|  2| val_2|
// |  4| val_4|  4| val_4|
// |  5| val_5|  5| val_5|
// ...
```

2. Java

```
importjava.io.Serializable;
importjava.util.ArrayList;
importjava.util.List;

importorg.apache.spark.api.java.function.MapFunction;
importorg.apache.spark.sql.Dataset;
importorg.apache.spark.sql.Encoders;
importorg.apache.spark.sql.Row;
importorg.apache.spark.sql.SparkSession;

publicstaticclassRecordimplementsSerializable{
privateintkey;
privateStringvalue;
```

```
publicintgetKey(){
returnkey;
}

publicvoidsetKey(intkey){
this.key=key;
}

publicStringgetValue(){
returnvalue;
}

publicvoidsetValue(Stringvalue){
this.value=value;
}
}

// warehouseLocation points to the default location for managed databases and
tables
StringwarehouseLocation="file:"+System.getProperty("user.dir")+"spark-
warehouse";
SparkSessionspark=SparkSession
.builder()
.appName("Java Spark Hive Example")
.config("spark.sql.warehouse.dir",warehouseLocation)
.enableHiveSupport()
.getOrCreate();

spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)");
spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO
TABLE src");

// Queries are expressed in HiveQL
spark.sql("SELECT * FROM src").show();
// +---+-------+
// |key|  value|
// +---+-------+
// |238|val_238|
// | 86| val_86|
```

```
// |311|val_311|
// ...

// Aggregation queries are also supported.
spark.sql("SELECT COUNT(*) FROM src").show();
// +--------+
// |count(1)|
// +--------+
// |    500 |
// +--------+

// The results of SQL queries are themselves DataFrames and support all normal
functions.
Dataset<Row>sqlDF=spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER
BY key");

// The items in DaraFrames are of type Row, which lets you to access each
column by ordinal.
Dataset<String>stringsDS=sqlDF.map(newMapFunction<Row,String>(){
@Override
publicStringcall(Rowrow)throwsException{
return"Key: "+row.get(0)+", Value: "+row.get(1);
}
},Encoders.STRING());
stringsDS.show();
// +--------------------+
// |               value|
// +--------------------+
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// ...

// You can also use DataFrames to create temporary views within a SparkSession.
List<Record>records=newArrayList<>();
for(intkey=1;key<100;key++){
Recordrecord=newRecord();
record.setKey(key);
record.setValue("val_"+key);
```

```
records.add(record);
}
Dataset<Row>recordsDF=spark.createDataFrame(records,Record.class);
recordsDF.createOrReplaceTempView("records");

// Queries can then join DataFrames data with data stored in Hive.
spark.sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show();
// +---+------+---+------+
// |key| value|key| value|
// +---+------+---+------+
// |  2| val_2|  2| val_2|
// |  2| val_2|  2| val_2|
// |  4| val_4|  4| val_4|
// ...
```

3. Python

```
fromos.pathimportexpanduser,join

frompyspark.sqlimportSparkSession
frompyspark.sqlimportRow

# warehouse_location points to the default location for managed databases and
tables
warehouse_location='file:${system:user.dir}/spark-warehouse'

spark=SparkSession \
.builder \
.appName("Python Spark SQL Hive integration example") \
.config("spark.sql.warehouse.dir",warehouse_location) \
.enableHiveSupport() \
.getOrCreate()

# spark is an existing SparkSession
spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO
TABLE src")

# Queries are expressed in HiveQL
spark.sql("SELECT * FROM src").show()
# +---+-------+
# |key|  value|
# +---+-------+
# |238|val_238|
# | 86| val 86|
# |311|val 311|
# ...
```

```
# Aggregation queries are also supported.
spark.sql("SELECT COUNT(*) FROM src").show()
# +--------+
# |count(1)|
# +--------+
# |    500 |
# +--------+

# The results of SQL queries are themselves DataFrames and support all normal
functions.
sqlDF=spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

# The items in DaraFrames are of type Row, which allows you to access each
column by ordinal.
stringsDS=sqlDF.rdd.map(lambdarow:"Key: %d, Value: %s"%(row.key,row.value))
forrecordinstringsDS.collect():
print(record)
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# ...

# You can also use DataFrames to create temporary views within a SparkSession.
Record=Row("key","value")
recordsDF=spark.createDataFrame(map(lambdai:Record(i,"val "+str(i)),range(1,10
1)))
recordsDF.createOrReplaceTempView("records")

# Queries can then join DataFrame data with data stored in Hive.
spark.sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show()
# +---+------+---+------+
# |key| value|key| value|
# +---+------+---+------+
# |  2| val_2|  2| val_2|
# |  4| val_4|  4| val_4|
# |  5| val_5|  5| val_5|
# ...
```

4. R

```
# enableHiveSupport defaults to TRUE
sparkR.session(enableHiveSupport =TRUE)
sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE
src")

# Queries can be expressed in HiveQL.
results <- collect(sql("FROM src SELECT key, value"))
```

关于和不同版本的 Hive 元存储交互。Spark SQL 对 Hive 支持的最重要的特点之一是与

Hive 元存储的交互，这使得 SparkSQL 可以访问 Hive 表的元数据。从 Spark1.4.0 开始，可以使用一个 Spark SQL 的二进制构建来查询不同版本的 Hive 元存储。Spark SQL 在内部编译 Hive1.2.1，并且使用这些 classes 用于内部执行（序列化、反序列化、UDFs、UDAFs 等）。

4.8.5　用 JDBC 连接其他数据库

Spark SQL 同样包括可以使用 JDBC 从其他数据库读取数据的数据源。此功能优先使用 JdbcRDD，这是因为返回的结果作为一个 DataFrame，并且可以轻松地使用 Spark SQL 处理或者与其他数据源进行连接。利用 Java 或者 Python 可以更容易地使用 JDBC 数据源，因为它们不需要用户提供 ClassTag。（注意这与 Spark SQLJDBC 服务器可以允许其他应用使用 Spark SQL 执行查询语句不同）

在开始之前你需要将你指定的数据库的 JDBC driver 包含在 Spark 的环境变量中。例如，为了从 Spark Shell 连接到 postgres，你需要执行以下命令：

```
SPARK_CLASSPATH=postgresql-9.3-1102-jdbc41.jar bin/spark-shell
```

远程数据库的表可以通过 Data Sources API，用 DataFrame 或者 Spark SQL 临时表来装载。如表 4-10 所示是其选项列表。

表 4-10　选项列表

属性名	含义
url	需要连接的 JDBC URL
dbtable	需要读取的 JDBC 表。注意，任何可以填在 SQL 的 where 子句中的东西都可以填在这里（既可以填完整的表名，也可填括号括起来的子查询语句）
driver	JDBC driver 的类名。这个类必须在 Master 和 Worker 节点上都可用，这样各个节点才能将 driver 注册到 JDBC 的子系统中
partitionColumn、lowerBound、upperBound、numPartitions	这几个选项，如果指定其中一个，则必须全部指定。他们描述了多个 Worker 如何并行地读入数据，并将表分区。partitionColumn 必须是所查询的表中的一个数值字段。注意，lowerBound 和 upperBound 只是用于决定分区跨度的，而不是过滤表中的行。因此，表中所有的行都会被分区，然后返回
fetchSize	JDBC fetch size，决定每次获取多少行数据。在 JDBC 驱动上设成较小的值有利于性能优化（如，Oracle 上设为 10）

```
val jdbcDF = sqlContext.read.format("jdbc").options(
  Map("url" ->"jdbc:postgresql:dbserver",
"dbtable" ->"schema.tablename")).load()
```

需要注意：JDBC driver class 必须在所有 client session 或者 executor 上，对 Java 的原生 classloader 可见。这是因为 Java 的 DriverManager 在打开一个连接之前，会做安全检查，并忽略所有对原声 classloader 不可见的 driver。最简单的一种方法就是在所有 Worker 节点上修改 compute_classpath.sh，并包含你所需的 driver jar 包。

一些数据库，如 H2，会把所有的名字转大写。对于这些数据库，在 Spark SQL 中必须也使用大写。

4.9 Spark 性能调优

对于有一定计算量的 Spark 作业来说，可能的性能改进的方式：不是把数据缓存在内存里，就是调整一些开销较大的选项参数。

1. 缓存数据到内存

Spark SQL 可以通过调用 spark.cacheTable("tableName")或 dataFrame.cache()来将表以列式形式缓存在内存中，然后 Spark SQL 可以只扫描需要的列，并且可以自动调节压缩以最小化内存使用率和 GC 压力。可以调用 spark.uncacheTable("tableName")来将表从内存中删除。

可以在 SparkSession 上使用 setConf 方法来配置内存缓存，或者使用 SQL 执行 SET key=value 命令。如表 4-11 所示为查询任务配置内存缓存方法。

表 4-11 查询任务配置内存缓存方法

属性名	默认值	含义
spark.sql.inMemoryColumnarStorage.compressed	true	如果设置为 true，Spark SQL 将会根据数据统计信息自动为每一列选择单独的压缩编码方式
spark.sql.inMemoryColumnarStorage.batchSize	10000	控制列式缓存批量的大小。增大批量大小可以提高内存利用率和压缩率，但同时也会带来 OOM（Out Of Memory）的风险

2. 其他配置选项

如表 4-12 所示的选项同样可以用于查询语句执行时的性能调优，在以后发布的版本中可能会弃用这些选项，更多地将优化改为自动执行。

表 4-12 查询任务配置内存缓存的其他方法

属性名	默认值	含义
spark.sql.autoBroadcastJoinThreshold	10485760 (10 MB)	配置 join 操作时，能够作为广播变量的最大 table 的大小。设置为-1，表示禁用广播。注意，目前的元数据统计仅支持 Hive metastore 中的表，并且需要运行这个命令：ANALYSE TABLE <tableName> COMPUTE STATISTICS noscan
spark.sql.tungsten.enabled	true	设为 true，则启用优化的 Tungsten 物理执行后端。Tungsten 会显式地管理内存，并动态生成表达式求值的字节码
spark.sql.shuffle.partitions	200	配置数据混洗（shuffle）时（join 或者聚合操作）使用的分区数

4.10　分布式 SQL 引擎

Spark SQL 同样可以使用 JDBC/ODBC 或者命令行接口来作为一个分布式查询引擎。在这种模式中，终端用户或者应用可以通过执行 SQL 查询语句直接与 Spark SQL 进行交互，不需要写任何代码。

1. 运行 Thrift JDBC/ODBC 服务

这里实现的 Thrift JDBC/ODBC server 和 Hive-1.2.1 中的 HiveServer2 是相同的。你可以使用 beeline 脚本来测试 Spark 或者 Hive-1.2.1 的 JDBC server。

在 Spark 目录下运行下面这个命令，启动一个 JDBC/ODBC server。

```
./sbin/start-thriftserver.sh
```

这个脚本能接受所有 bin/spark-submit 命令支持的选项参数，外加一个 –hiveconf 选项来指定 Hive 属性，运行./sbin/start-thriftserver.sh –help 可以查看完整的选项列表。默认情况下，启动的 server 将会在 localhost:10000 端口上监听。要改变监听主机名或端口，可以用以下环境变量：

```
export HIVE_SERVER2_THRIFT_PORT=<listening-port>
export HIVE_SERVER2_THRIFT_BIND_HOST=<listening-host>
./sbin/start-thriftserver.sh \
  --master <master-uri> \
  ...
```

或者 Hive 系统属性来指定：

```
./sbin/start-thriftserver.sh \
  --hiveconf hive.server2.thrift.port=<listening-port> \
  --hiveconf hive.server2.thrift.bind.host=<listening-host> \
  --master <master-uri>
  ...
```

接下来，可以开始在 beeline 中测试这个 Thrift JDBC/ODBC server：

```
./bin/beeline
```

下面的指令，可以连接到一个 JDBC/ODBC server：

```
beeline> !connect jdbc:hive2://localhost:10000
```

Hive 的配置是在 conf/目录下的 hive-site.xml、core-site.xml、hdfs-site.xml 中指定的，可以在 beeline 的脚本中指定。

Thrift JDBC server 也支持通过 HTTP 传输 Thrift RPC 消息，以下配置（在 conf/hive-

site.xml 中）将启用 HTTP 模式：

```
hive.server2.transport.mode - Set this to value: http
hive.server2.thrift.http.port - HTTP port number fo listen on; default is
10001
hive.server2.http.endpoint - HTTP endpoint; default is cliservice
```

同样，在 beeline 中也可以用 HTTP 模式连接 JDBC/ODBC server：

```
beeline> !connect jdbc:hive2://<host>:<port>/<database>?hive.server2.transport.
mode=http;hive.server2.thrift.http.path=<http_endpoint>
```

2. 使用 Spark SQL 命令行工具 CLI

Spark SQL CLI 是一个很方便的工具，它可以用 local mode 运行 hive metastore service，并且在命令行中执行输入的查询。注意 Spark SQL CLI 目前还不支持和 Thrift JDBC server 通信。使用如下命令，在 spark 目录下启动一个 Spark SQL CLI：

```
./bin/spark-sql
```

Hive 配置在 conf 目录下 hive-site.xml、core-site.xml、hdfs-site.xml 中设置，可以用这个命令查看完整的选项列表：

```
./bin/spark-sql -help
```

4.11 本章小结

Apache Spark 是一个围绕速度、易用性和复杂分析构建的大数据处理框架。Spark 运行在现有的 Hadoop 分布式文件系统（HDFS）基础之上，提供额外的增强功能。它支持将 Spark 应用部署到现存的 Hadoop v1 集群（with SIMR — Spark Inside MapReduce）或 Hadoop v2 YARN 集群甚至是 Apache Mesos 之中，提供一个管理不同的大数据用例和需求的全面且统一的解决方案。本章主要介绍 Spark 通过全面、统一的框架管理各种有着不同性质（文本数据、图表数据等）的数据集和数据源（批量数据或实时的流数据）的大数据处理的需求，同时例举基于 Spark 用 Java、R、Scala 或 Python 快速的程序编写，最后介绍 Spark SQL 查询和 DataFrame 分布式数据集数据处理。

第 5 章

Spark MLlib机器学习算法实现

Spark 之所以在机器学习方面具有得天独厚的优势，有以下原因[55]：

（1）机器学习算法一般都有很多个步骤迭代计算的过程，机器学习的计算需要在多次迭代后获得足够小的误差或者足够收敛才会停止，迭代时如果使用 Hadoop 的 MapReduce 计算框架，每次计算都要读/写磁盘以及任务的启动等工作，这会导致非常大的 I/O 和 CPU 消耗。而 Spark 基于内存的计算模型天生就擅长迭代计算，多个步骤计算直接在内存中完成，只有在必要时才会操作磁盘和网络，所以说 Spark 正是机器学习的理想的平台。

（2）从通信的角度讲，如果使用 Hadoop 的 MapReduce 计算框架，JobTracker 和 TaskTracker 之间由于是通过 heartbeat 的方式来进行的通信和传递数据，会导致非常慢的执行速度，而 Spark 具有出色而高效的 Akka 和 Netty 通信系统，通信效率极高[56]。

5.1 Spark MLlib 基础

MLlib（Machine Learnig lib）是 Spark 对常用的机器学习算法的实现库，同时包括相关的测试和数据生成器。Spark 的设计初衷就是为了支持一些迭代的 Job，这正好符合很多机器学习算法的特点[57-65]。MLlib 目前支持 4 种常见的机器学习问题：分类、回归、聚类和协同过滤，MLlib 在 Spark 整个生态系统中的位置如图 5-1 所示。

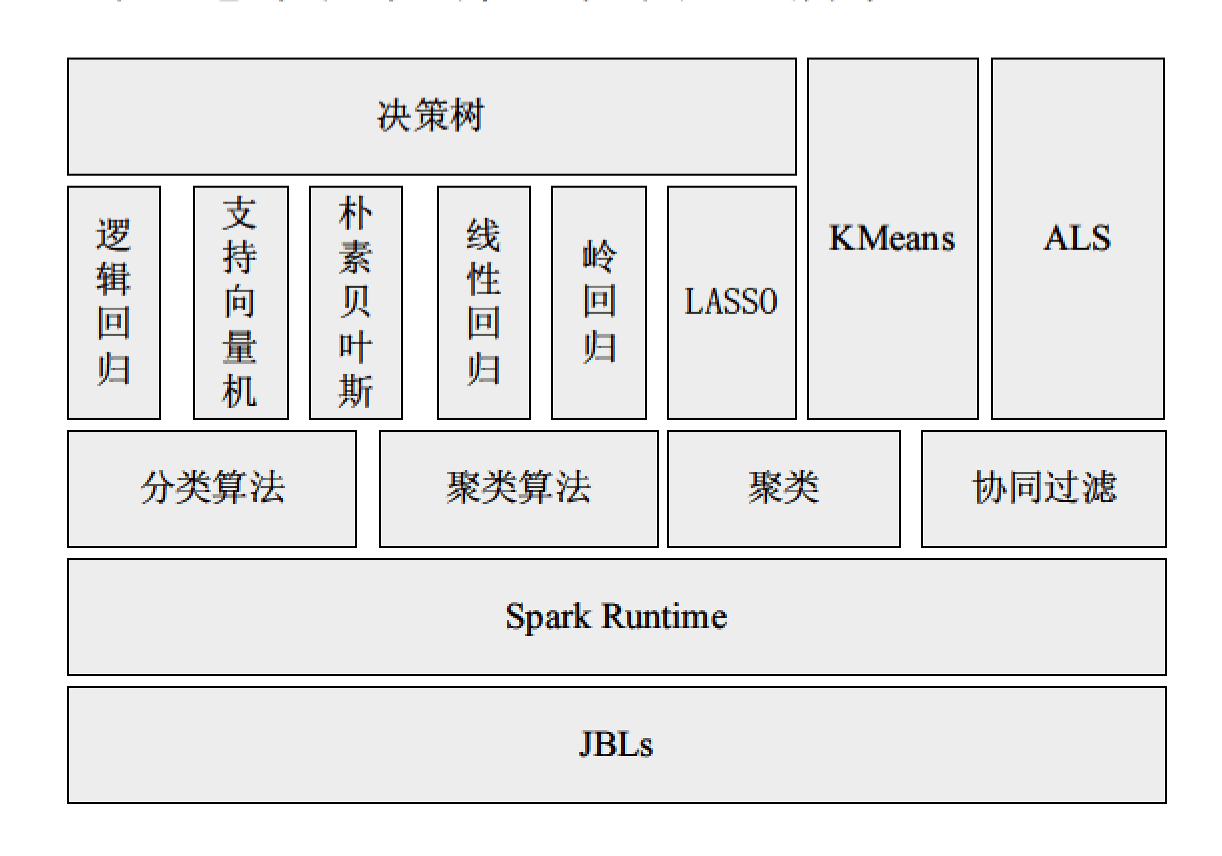


图 5-1　MLlib 在 Spark 整个生态系统中的位置

5.1.1 机器学习

机器学习（Machine Learning，ML）是一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多门学科。专门研究计算机怎样模拟或实现人类的学习行为，以获取新的知识或技能，重新组织已有的知识结构使之不断改善自身的性能。它是人工智能的核心，是使计算机具有智能的根本途径，其应用遍及人工智能的各个领域，它主要使用归纳、综合，而不是演绎。

Tom Mitchell 的机器学习（1997）对信息论中的一些概念有详细的解释，其中定义机器学习时提到，“机器学习是对能通过经验自动改进的计算机算法的研究。”（Machine Learning is the study of computer algorithms that improve automatically through experience.）。

Alpaydin（2004）同时提出自己对机器学习的定义，“机器学习是用数据或以往的经验，以此优化计算机程序的性能标准。”（Machine learning is programming computers to optimize a performance criterion using example data or past experience.）。

严格的提法是：机器学习是一门研究机器获取新知识和新技能，并识别现有知识的学问。这里所说的“机器”，指的就是计算机，包括电子计算机、中子计算机、光子计算机或神经计算机等。

机器学习已经有了十分广泛的应用，例如：数据挖掘、计算机视觉、自然语言处理、生物特征识别、搜索引擎、医学诊断、检测信用卡欺诈、证券市场分析、DNA 序列测序、语音和手写识别、战略游戏和机器人运用。

5.1.2 机器学习分类

机器学习是数据通过算法构建出模型并对模型进行评估，评估的性能如果达到要求就拿这个模型来测试其他的数据，如果达不到要求就要调整算法来重新建立模型，再次进行评估，如此循环往复，最终获得满意的经验来处理其他的数据。

1. 监督学习

监督是从给定的训练数据集中学习一个函数（模型），当新的数据到来时，可以根据这个函数（模型）预测结果。监督学习的训练集要求包括输入和输出，也可以说是特征和目标。训练集中的目标是由人标注（标量）的。在监督式学习下，输入数据被称为“训练数据”，每组训练数据有一个明确的标识或结果，如对防垃圾邮件系统中“垃圾邮件”“非垃圾邮件”，对手写数字识别中的“1”“2”“3”等。在建立预测模型时，监督式学习建立一个学习过程，将预测结果与“训练数据”的实际结果进行比较，不断调整预测模型，直到模型的预测结果达到一个预期的准确率。常见的监督学习算法包括回归分析和统计分类。

监督学习常常用于分类，因为目标往往是让计算机去学习我们已经创建好的分类系统。数字识别再一次成为分类学习的常见样本。一般来说，对于那些有用的分类系统和容易判断的分类系统，分类学习都适用。

（1）二元分类是机器学习要解决的基本问题，将测试数据分成两个类，如垃圾邮件的判别、房贷是否允许等问题的判断。

（2）多元分类是二元分类的逻辑延伸。例如，在因特网的流分类的情况下，根据问题的分类，网页可以被归类为体育、新闻、技术等，依此类推。

监督学习是训练神经网络和决策树的最常见技术。神经网络和决策树技术高度依赖于事先确定的分类系统给出的信息。对于神经网络来说，分类系统用于判断网络的错误，然后调整网络去适应它；对于决策树，分类系统用来判断哪些属性提供了最多的信息，如此一来可以用它解决分类系统的问题。

2. 半监督学习

半监督学习（Semi-supervised Learning）是介于监督学习与无监督学习之间的一种机器学习方式，是模式识别和机器学习领域研究的重点问题。它主要考虑如何利用少量的标注样本和大量的未标注样本进行训练和分类的问题。半监督学习对于减少标注代价，提高学习机器性能具有非常重大的实际意义。主要算法有五类：基于概率的算法；在现有监督算法基础上进行修改的方法；直接依赖于聚类假设的方法等，在此学习方式下，输入数据部分被标识，部分没有被标识，这种学习模型可以用来进行预测，但是模型首先需要学习数据的内在结构以便合理地组织数据来进行预测。应用场景包括分类和回归，算法包括一些对常用监督式学习算法的延伸，这些算法首先试图对未标识数据进行建模，在此基础上再对标识的数据进行预测，如图论推理算法（Graph Inference）或者拉普拉斯支持向量机（Laplacian SVM）等。

半监督学习分类算法提出的时间比较短，还有许多方面没有更深入地研究。半监督学习从诞生以来，主要用于处理人工合成数据，无噪声干扰的样本数据是当前大部分半监督学习方法使用的数据，而在实际生活中用到的数据却大部分不是无干扰的，通常都比较难以得到纯样本数据。

3. 强化学习

强化学习通过观察来学习动作的完成，每个动作都会对环境有所影响，学习对象根据观察到的周围环境的反馈来做出判断。在这种学习模式下，输入数据作为对模型的反馈，不像监督模型那样，输入数据仅仅是作为一个检查模型对错的方式。在强化学习下，输入数据直接反馈到模型，模型必须对此立刻做出调整。常见的应用场景包括动态系统以及机器人控制等。常见算法包括 Q-Learning 以及时间差学习（Temporal difference learning）。

在企业数据应用的场景下，人们最常用的可能就是监督式学习和非监督学习的模型。在图像识别等领域，由于存在大量的非标识的数据和少量的可标识数据，所以半监督学习是一个很热的话题；而强化学习更多地应用在机器人控制及其他需要进行系统控制的领域。

5.1.3 机器学习常见算法

根据算法的功能和形式的类似性，我们可以把算法分类，比如说基于树的算法、基于神经网络的算法等。当然，机器学习的范围非常庞大，有些算法很难明确归类到某一类；而对于有些分类来说，同一分类的算法可以针对不同类型的问题。下面用一些相对比较容易理解的方式来解析一些主要的机器学习算法：

1. 回归算法

回归算法是试图采用对误差的衡量来探索变量之间的关系的一类算法。回归算法是统计机器学习的利器。在机器学习领域，人们说起回归，有时候是指一类问题，有时候是指一类算法，这一点常常会使初学者有所困惑。常见的回归算法包括：最小二乘法（Ordinary Least Square）、逻辑回归（Logistic Regression）、逐步式回归（Stepwise Regression）、多元自适应回归样条（Multivariate Adaptive Regression Splines）以及本地散点平滑估计（Locally Estimated Scatterplot Smoothing）等。

2. 基于实例的算法

基于实例的算法常常用来对决策问题建立模型，这样的模型常常先选取一批样本数据，然后根据某些近似性把新数据与样本数据进行比较，通过这种方式来寻找最佳的匹配。因此，基于实例的算法常常也被称为“赢家通吃”学习或者“基于记忆的学习”。常见的算法包括 k-Nearest Neighbor （KNN）、学习矢量量化（Learning Vector Quantization， LVQ）以及自组织映射算法（Self-Organizing Map，SOM）等。

3. 正则化方法

正则化方法是其他算法（通常是回归算法）的延伸，根据算法的复杂度对算法进行调整。正则化方法通常对简单模型予以奖励而对复杂算法予以惩罚。常见的算法包括：Ridge Regression、Least Absolute Shrinkage and Selection Operator（LASSO）以及弹性网络（Elastic Net）等。

4. 决策树学习

决策树算法根据数据的属性采用树状结构建立决策模型，决策树模型常常用来解决分类和回归问题。常见的算法包括：分类及回归树（Classification And Regression Tree， CART）、ID3 （Iterative Dichotomiser 3）、C4.5、Chi-squared Automatic Interaction Detection （CHAID）、Decision Stump、随机森林（Random Forest）、多元自适应回归样条（MARS）以及梯度推进机（Gradient Boosting Machine，GBM）等。

5. 贝叶斯学习

贝叶斯算法是基于贝叶斯定理的一类算法，主要用来解决分类和回归问题。常见算法包括：朴素贝叶斯算法、平均单依赖估计（Averaged One-Dependence Estimators， AODE）以及 Bayesian Belief Network（BBN）等。

6. 基于核的算法

基于核的算法中最著名的莫过于支持向量机（SVM）了。基于核的算法把输入数据映射到一个高阶的向量空间， 在这些高阶向量空间里，有些分类或者回归问题能够更容易解决。常见的基于核的算法包括：支持向量机（Support Vector Machine，SVM）、径向基函数（Radial Basis Function，RBF）以及线性判别分析（Linear Discriminate Analysis，LDA）等。

7. 聚类算法

聚类就像回归一样，有时候人们描述的是一类问题，有时候描述的是一类算法。聚类算法通常按照中心点或者分层的方式对输入数据进行归并。所有的聚类算法都试图找到数据的内在结构，以便按照最大的共同点将数据进行归类。常见的聚类算法包括 k-Means 算法以及期望最大化算法（Expectation Maximization，EM）。

8. 关联规则学习

关联规则学习通过寻找最能够解释数据变量之间关系的规则，来找出大量多元数据集中有用的关联规则。常见算法包括 Apriori 算法和 Eclat 算法等。

9. 人工神经网络算法

人工神经网络算法模拟生物神经网络，是一类模式匹配算法。通常用于解决分类和回归问题。人工神经网络是机器学习的一个庞大的分支，有几百种不同的算法（其中深度学习就是其中的一类算法，我们会单独讨论）。重要的人工神经网络算法包括：感知器神经网络（Perceptron Neural Network）、反向传递（Back Propagation）、Hopfield 网络、自组织映射（Self-Organizing Map, SOM）、学习矢量量化（Learning Vector Quantization，LVQ）等。

10. 深度学习算法

深度学习算法是对人工神经网络的发展，在近期赢得了很多关注，特别是百度也开始发力深度学习后，更是在国内引起了很多关注。在计算能力变得日益廉价的今天，深度学习试图建立大得多也复杂得多的神经网络。很多深度学习的算法是半监督式学习算法，用来处理存在少量未标识数据的大数据集。常见的深度学习算法包括：受限玻尔兹曼机（Restricted Boltzmann Machine，RBN）、Deep Belief Networks（DBN）、卷积网络（Convolutional Network）、堆栈式自动编码器（Stacked Auto-encoders）等。

11. 降低维度算法

像聚类算法一样，降低维度算法试图分析数据的内在结构，不过降低维度算法是以非监督学习的方式，试图利用较少的信息来归纳或者解释数据。这类算法可以用于高维数据的可视化或者用来简化数据以便监督式学习使用。常见的算法包括：主成分分析（Principle Component Analysis，PCA）、偏最小二乘回归（Partial Least Square Regression，PLS）、Sammon 映射、多维尺度（Multi-Dimensional Scaling, MDS）、投影追踪（Projection Pursuit）等。

12. 集成算法

集成算法用一些相对较弱的学习模型独立地对同样的样本进行训练，然后把结果整合起来进行整体预测。集成算法的主要难点在于究竟集成哪些独立的、较弱的学习模型以及如何把学习结果整合起来。这是一类非常强大的算法，同时也非常流行。常见的算法包括：Boosting、Bootstrapped Aggregation（Bagging）、AdaBoost、堆叠泛化（Stacked Generalization，Blending）、梯度推进机（Gradient Boosting Machine, GBM）、随机森林

（Random Forest）等。

5.1.4　Spark MLlib 机器学习库

MLlib 基于 RDD，可以与 Spark SQL、GraphX、Spark Streaming 无缝集成，以 RDD 为基石，4 个子框架可构建大数据计算中心。

MLlib 是 MLBase 的一部分，其中 MLBase 分为 4 部分：MLlib、MLI、ML Optimizer 和 MLRuntime。

- ML Optimizer 会选择它认为最适合的、已经在内部实现好了的机器学习算法和相关参数，来处理用户输入的数据，并返回模型或别的帮助分析的结果。
- MLI 是一个进行特征抽取和高级 ML 编程抽象的算法实现的 API 或平台。
- MLlib 是 Spark 实现一些常见的机器学习算法和实用程序，包括分类、回归、聚类、协同过滤、降维以及底层优化，该算法可以进行可扩充。
- MLRuntime 基于 Spark 计算框架，将 Spark 的分布式计算应用到机器学习领域。

1. Spark MLlib 架构解析

图 5-2 给出了 Spark MLlib 架构解析，从架构图可以看出 MLlib 主要包含三个组成部分：

- 底层基础：包括 Spark 的运行库、矩阵库和向量库。
- 算法库：包含广义线性模型、推荐系统、聚类、决策树和评估的算法。
- 实用程序：包括测试数据的生成、外部数据的读入等功能。

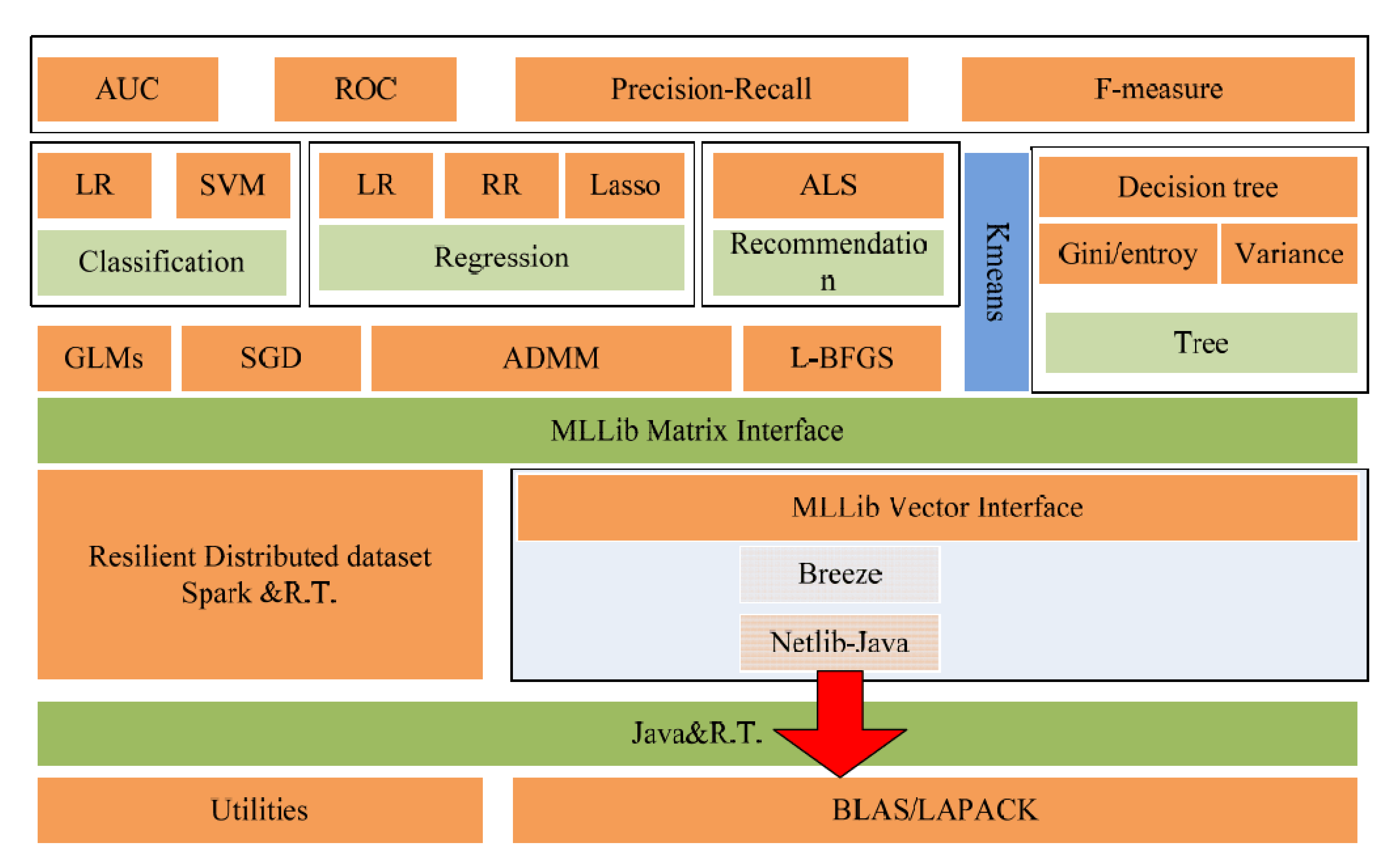


图 5-2　Spark MLlib 架构解析

2. MLlib 的底层基础解析

底层基础部分主要包括向量接口和矩阵接口，这两种接口都会使用 Scala 语言基于 Netlib 和 BLAS/LAPACK 开发的线性代数库 Breeze。

MLlib 支持本地的密集向量和稀疏向量，并且支持标量向量。

MLlib 同时支持本地矩阵和分布式矩阵，支持的分布式矩阵分为 RowMatrix、IndexedRowMatrix、CoordinateMatrix 等。

关于密集型和稀疏型的向量 Vector 的示例如下所示：

```
Importorg.apache.spark.mllib.linalg.{vector,vectors}
//Create a dense vector(1.0,0.0,3.0).
Val dv:vector=vectors.dense(1.0,0.0,3.0)
//Create a sparse vector(1.0,0.0,3.0)by specifying its indices and values
corresponding to nonzero entries.
Val sv1:vector=vectors.sparse(3,Array(0,2),Array(1.0,3.0))
//Create a sparse vector(1.0,0.0,3.0)by specifying its nonzero entries.
Val sv2:vector=vectors.sparse(3,Seq((0,1.0),(2,3.0)))
```

$$dense:\underline{1.0.0.0.0.0.3.}$$

$$sparse:\begin{cases} size:7 \\ indices:\underline{0}\ \underline{6} \\ values:\underline{1.3.} \end{cases}$$

稀疏矩阵在含有大量非零元素的向量 Vector 计算中，会节省大量的空间并大幅度提高计算速度，示例如下所示：

Training set	
12 million	examples
500	features
sparsity:	10%

	dense	sparse
storage	47G	7G
time	240s	58s

标量 LabledPoint 在实际中也被大量使用，例如判断邮件是否为垃圾邮件时就可以使用类似于以下的代码：

```
Import org.apache.spark.mllib.linalg.vectors
Import org.apache.spark.mllib.regression.LabeledPoint
//Create a labeled point with a position label and a dense feature vector.
Val pos=LabeledPoint(1.0,Vectors.dense(1.0,0.0,3.0))
//Create a labeled point with a negative label and a sparse feature vector.
Val neg=LabeledPoint(0.0,Vectors.sparse(3,Array(0,2),Array(1.0,3.0)))
```

可以把表示为 1.0 的判断为正常邮件，而表示为 0.0 则作为垃圾邮件来看待。

对于矩阵 Matrix 而言，本地模式的矩阵如下所示：

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{pmatrix}$$

```
Import org.apache.spark.mllib.linalg.{Matrix,Matrices}
//Create a dense matrix(1.0,2.0),(3.0,4.0),(5.0,6.0))
Valdm:Matrix=Matrices.dense(3,2,Array(1.0,3.0,5.0,2.0,4.0,6.0))
```

分布式矩阵如图 5-3 所示：

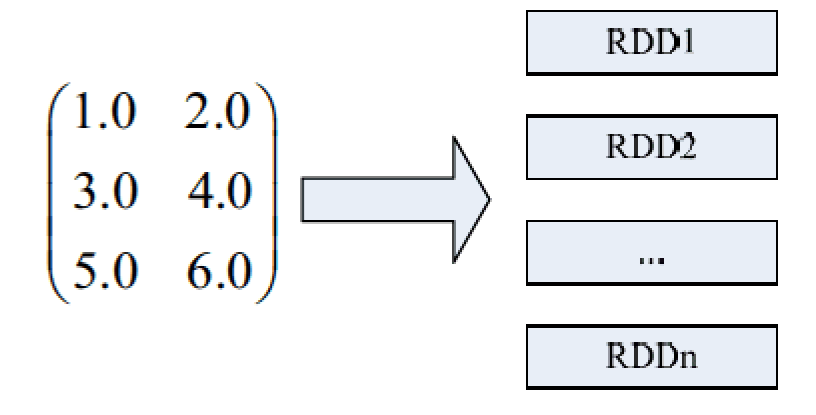


图 5-3　分布式矩阵

RowMatrix 直接通过 RDD[Vector]来定义并可以用来统计平均数、方差、协同方差等：

```
Import org.apache.spark.mllib.linalg.Vector
Import org.apache.spark.mllib.linalg.distributed.RowMatrix
Valrow:RDD[vector]=…//an RDD of local vectors
//Create a RowMatrix from an RDD[vector].
Valmat:RowMatrix=new RowMatrix(rows)
//Get its size.
Val m=mat.numRows()
Val n=mat.numCols()
Import org.apache.spark.mllib.linalg.Matrix
Import org.apache.spark.mllib.linalg.distributed.RowMatirx
Import org.apache.spark.mllib.stat.MultivariatesStatisticalSummary

Val mat:RowMatirx=…//a RowMatirx

//compute column summary statistics.
Val summary:multivariateStatisticalSummary=mat.computColunmSummarySatistics()
printLn(summary.mean)//a dense vector containing the mean value for each colum
prinLn(summary.mean)//column-wise variance
printLn(summary.numNonzeros)//number of nonzeros in each column
//Compute the covariance matrix
Val cov:Matrix=mat.computeCovariance()
```

而 IndexedRowMatrix 是带有索引的 Matrix，但其可以通过 toRowMatrix 方法来转换为 RowMatrix，从而利用其统计功能，代码示例如下所示：

```
Import org.apache.spark.mllib.linalg.distributed.{IndexedRow,IndexedRow
Matrix,RowMatrix}
Val rows:RDD[IndexedRow]=…//an RDD of index rows
//Create an IndexedRowMatrix from an RDD[IndexedRow].
Val mat:IndexedRowMatrix=new IndexedRowMatrix(rows)
//get its size.
Val m=mat.numRow()
Val n=mat.numCols()
//drop its row indices.
Val rowMat:RowMatrix=mat.toRowMatrix()
```

CoordinateMatrix 常用于稀疏性比较高的计算中，是由 RDD[MatrixEntry]来构建的，MatrixEntry 是一个 Tuple 类型的元素，其中包含行、列和元素值，代码示例如下所示：

```
Import  org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix,MatrixEntry}
Val entries:RDD[MatrixEntry]=…//an Rdd of matrix entries
//create a coordinatematrix from an RDD[MatrixEntry].
Val mat:CoordinateMatrix=new CoordinateMatrix(entries)
//get its size
Val m=mat.numRows()
Val n=mat.numCols()
//convert it to an IndexRowMatrix whose rows are sparse vectors.
ValindexRowMatrix=mat.toIndexedRowMatrix()
```

3. MLlib 的算法库核心

如图 5-4 所示的是 MLlib 算法库的核心内容。

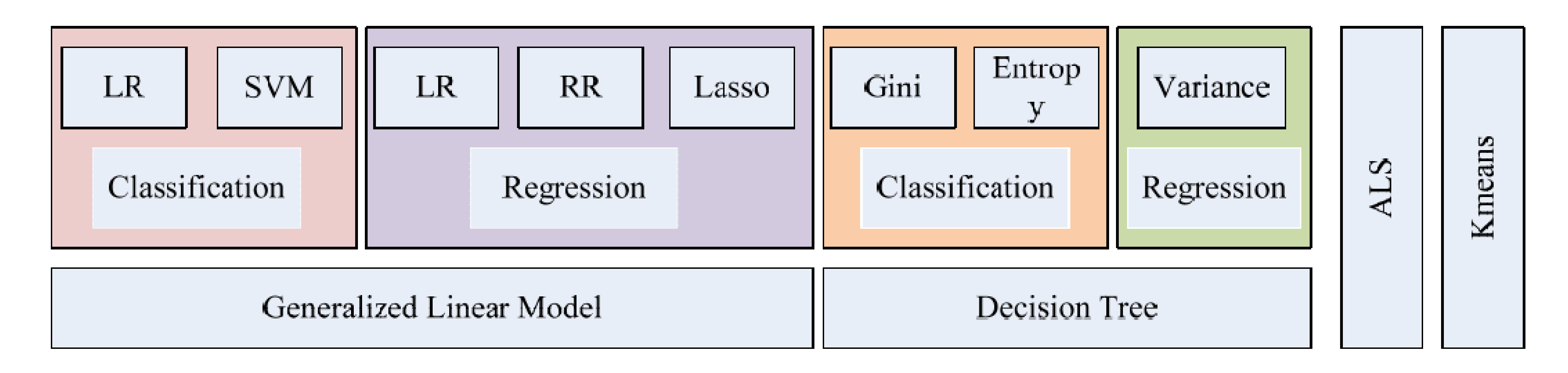


图 5-4 MLlib 算法库的核心内容

5.1.5 基于 Spark 常用的算法举例分析

1. 分类算法

分类算法属于监督式学习，使用类标签已知的样本建立一个分类函数或分类模型，应用分类模型，能把数据库中的类标签未知的数据进行归类。分类在数据挖掘中是一项重要的任务，目前在商业上应用最多，常见的典型应用场景有流失预测、精确营销、客户获取、个性偏好等。MLlib 目前支持分类算法有：逻辑回归、支持向量机、朴素贝叶斯和决策树。

Spark 分类算法案例：导入训练数据集，然后在训练集上执行训练算法，最后在所得模型上进行预测并计算训练误差。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.classification.SVMWithSGD
import org.apache.spark.mllib.regression.LabeledPoint

//加载和解析数据文件
val data = sc.textFile("mllib/data/sample_svm_data.txt")
val parsedData = data.map { line =>
val parts = line.split(' ')
    LabeledPoint(parts(0).toDouble, parts.tail.map(x => x.toDouble).toArray)
}

//设置迭代次数并进行训练
val numIterations = 20
val model = SVMWithSGD.train(parsedData, numIterations)

//统计分类错误的样本比例
val labelAndPreds = parsedData.map { point =>
val prediction = model.predict(point.features)
(point.label, prediction)
}
val trainErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble /
parsedData.count
println("Training Error = " + trainErr)
```

2. 回归算法

回归算法属于监督式学习，每个个体都有一个与之相关联的实数标签，并且我们希望在给出用于表示这些实体的数值特征后，所预测出的标签值可以尽可能接近实际值。MLlib 目前支持回归算法有：线性回归、岭回归、Lasso 和决策树。

Spark 回归算法案例：导入训练数据集，将其解析为带标签点的 RDD，使

用 LinearRegressionWithSGD 算法建立一个简单的线性模型来预测标签的值，最后计算均方差来评估预测值与实际值的吻合度。

```
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.regression.LabeledPoint

//加载和解析数据文件
val data = sc.textFile("mllib/data/ridge-data/lpsa.data")
val parsedData = data.map { line =>
  val parts = line.split(',')
  LabeledPoint(parts(0).toDouble, parts(1).split(' ').map(x =>
x.toDouble).toArray)
}

//设置迭代次数并进行训练
val numIterations = 20
val model = LinearRegressionWithSGD.train(parsedData, numIterations)

//统计回归错误的样本比例
val valuesAndPreds = parsedData.map { point =>
val prediction = model.predict(point.features)
(point.label, prediction)
}
val MSE = valuesAndPreds.map{ case(v, p) => math.pow((v - p), 2)}.reduce(_ +
_)/valuesAndPreds.count
println("training Mean Squared Error = " + MSE)
```

3. 聚类算法

聚类算法属于非监督式学习，通常被用于探索性的分析，是根据“物以类聚”的原理，将本身没有类别的样本聚集成不同的组，这样的一组数据对象的集合叫做簇，并且对每一个这样的簇进行描述的过程。它的目的是使得属于同一簇的样本之间应该彼此相似，而不同簇的样本应该足够不相似，常见的典型应用场景有客户细分、客户研究、市场细分、价值评估。MLlib 目前支持广泛使用的 KMmeans 聚类算法。

Spark 聚类算法案例：导入训练数据集，使用 KMeans 对象来将数据聚类到两个类簇当中，所需的类簇个数会被传递到算法中，然后计算集内均方差总和 （WSSSE），可以通过增加类簇的个数 k 来减小误差。 实际上，最优的类簇数通常是 1，因为这一点通常是 WSSSE 图中的 “低谷点”。

```
importorg.apache.spark.mllib.clustering.KMeans
```

```
//加载和解析数据文件
val data = sc.textFile("kmeans_data.txt")
val parsedData = data.map( _.split(' ').map(_.toDouble))
//设置迭代次数、类簇的个数
val numIterations = 20
val numClusters = 2

//进行训练
val clusters = KMeans.train(parsedData, numClusters, numIterations)

//统计聚类错误的样本比例
val WSSSE = clusters.computeCost(parsedData)
println("Within Set Sum of Squared Errors = " + WSSSE)
```

4. 协同过滤

协同过滤常被应用于推荐系统，这些技术旨在补充用户-商品关联矩阵中所缺失的部分。MLlib 当前支持基于模型的协同过滤，其中用户和商品通过一小组隐语义因子进行表达，并且这些因子也用于预测缺失的元素。

Spark 协同过滤算法案例：导入训练数据集，数据每一行由一个用户、一个商品和相应的评分组成。假设评分是显性的，使用默认的 ALS.train()方法，通过计算预测出的评分的均方差来评估这个推荐模型。

```
import org.apache.spark.mllib.recommendation.ALS
import org.apache.spark.mllib.recommendation.Rating

//加载和解析数据文件
val data = sc.textFile("mllib/data/als/test.data")
val ratings = data.map(_.split(',') match {
case Array(user, item, rate) => Rating(user.toInt, item.toInt, rate.toDouble)
})

//设置迭代次数
val numIterations = 20
val model = ALS.train(ratings, 1, 20, 0.01)

//对推荐模型进行评分
val usersProducts = ratings.map{ case Rating(user, product, rate) => (user,
product)}
val predictions = model.predict(usersProducts).map{
case Rating(user, product, rate) => ((user, product), rate)
}
```

```
val ratesAndPreds = ratings.map{
case Rating(user, product, rate) => ((user, product), rate)
}.join(predictions)
val MSE = ratesAndPreds.map{
case ((user, product), (r1, r2)) => math.pow((r1- r2), 2)
}.reduce(_ + _)/ratesAndPreds.count
println("Mean Squared Error = " + MSE)
```

5.2 Spark MLlib 矩阵向量

Spark MLlib 底层的向量、矩阵运算使用了 Breeze 库，Breeze 库提供了 Vector/Matrix 的实现以及相应计算的接口（Linalg）。在 MLlib 里面同时也提供了 Vector 和 Linalg 等的实现[66-68]。

5.2.1 Breeze 创建函数

在使用 Breeze 库时，需要导入相关包：

```
import breeze.linalg._
import breeze.numerics._
```

API 参看 http://www.scalanlp.org/api/breeze/index.html#breeze.linalg.package。Breeze 创建函数操作如表 5-1 所示。

表 5-1　Breeze 创建函数

操作名称	Breeze 函数	输出结果	对应 Numpy 函数
全 0 矩阵	DenseMatrix.zeros[Double](2,3)	0.0 0.0 0.0 0.0 0.0 0.0	zeros((2,3))
全 0 向量	DenseVector.zeros[Double](3)	DenseVector(0.0,0.0,0.0)	zeros(3)
全 1 向量	DenseVector.ones[Double](3)	DenseVector(1.0,1.0,1.0)	ones(3)
按数值填充向量	DenseVector.fill(3){1.0}	DenseVector(1.0,1.0,1.0)	ones(3)*1.0
生成随机向量	DenseVector.range(start,end,step), Vector.rangeD(start,end,step)	DenseVector(1,3,5,7,9)	
线性等分向量（用于产生 Start 和 end 之间的 N 点行矢量）	DenseVector.linspace(start,end,numvals)		
单位矩阵	DenseMatr.eye[Double](3)	1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0	eye(3)
对角矩阵	Diag(DenseVector(1.0,2.0,3.0))	1.0 0.0 0.0 0.0 2.0 0.0 0.0 0.0 3.0	diag((1.0,2.0,3.0))

（续表）

操作名称	Breeze 函数	输出结果	对应 Numpy 函数
按照行创建矩阵	DenseMatrix((1.0,2.0),(3.0,4.0))	1.0 2.0 3.0 4.0	array([[1.0,2.0],[3.0,4.0]])
按照行创建向量	DenseVector(1,2,3,4)	[1 2 3 4]	array([1,2,3,4])
向量转置	DenseVector(1,2,3,4).t	$[1\ 2\ 3\ 4]^T$	array([1 2 3 4]).reshape(-1,1)
从函数创建向量	DenseVector.tabulate(3){i => i*2}	[0 1 4]	
从函数创建矩阵	DenseMatrix.tabulate(3,2){case(i,j) => i+j}	0 1 1 2 2 3	
从数组创建向量	new DenseVector(array(1, 2, 3,4))	[1 2 3 4]	
从数组创建矩阵	new DenseMatrix(2,3,array(11,12,13,21.22,23))	11 12 13 21 22 23	
0 到 1 的随机向量	DenseVector.rand(4)	[0.0222 0.2231 0.5356 0.6902]	
0 到 1 的随机矩阵	DenseMatrix.rand(2,3)	0.2122 0.3033 0.8675 0.6628 0.0023 0.9987	

```
scala> val m1 = DenseMatrix.zeros[Double](2,3)
m1: breeze.linalg.DenseMatrix[Double] =
0.0 0.0 0.0
0.0 0.0 0.0
scala> val v1 = DenseVector.zeros[Double](3)
v1: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.0, 0.0)
scala> val v2 = DenseVector.ones[Double](3)
v2: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 1.0, 1.0)
scala> val v3 = DenseVector.fill(3){5.0}
v3: breeze.linalg.DenseVector[Double] = DenseVector(5.0, 5.0, 5.0)
scala> val v4 = DenseVector.range(1,10,2)
v4: breeze.linalg.DenseVector[Int] = DenseVector(1, 3, 5, 7, 9)
scala> val m2 = DenseMatrix.eye[Double](3)
m2: breeze.linalg.DenseMatrix[Double] =
1.0 0.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
scala> val v6 = diag(DenseVector(1.0,2.0,3.0))
v6: breeze.linalg.DenseMatrix[Double] =
1.0 0.0 0.0
0.0 2.0 0.0
```

```
0.0 0.0 3.0
scala> val v8 = DenseVector(1,2,3,4)
v8: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4)
scala> val v9 = DenseVector(1,2,3,4).t
v9: breeze.linalg.Transpose[breeze.linalg.DenseVector[Int]] =
Transpose(DenseVector(1, 2, 3, 4))
scala> val v10 = DenseVector.tabulate(3){i => 2*i}
v10: breeze.linalg.DenseVector[Int] = DenseVector(0, 2, 4)
scala> val m4 = DenseMatrix.tabulate(3, 2){case (i, j) => i+j}
m4: breeze.linalg.DenseMatrix[Int] =
0 1
1 2
2 3
scala> val v11 = new DenseVector(Array(1, 2, 3, 4))
v11: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4)
scala> val m5 = new DenseMatrix(2, 3, Array(11, 12, 13, 21, 22, 23))
m5: breeze.linalg.DenseMatrix[Int] =
11 13 22
12 21 23
scala> val v12 = DenseVector.rand(4)
v12: breeze.linalg.DenseVector[Double] = DenseVector(0.7517657487447951,
0.8171495400874123, 0.8923542318540489, 0.174311259949119)
scala> val m6 = DenseMatrix.rand(2, 3)
m6: breeze.linalg.DenseMatrix[Double] =
0.5349430131148125 0.8822136832272578 0.7946323804433382
0.41097756311601086 0.3181490074596882 0.34195102205697414
```

5.2.2　Breeze 元素访问

Breeze 元素访问如表 5-2 所示。

表 5-2　Breeze 元素访问

操作名称	Breeze 函数	对应 Numpy 函数
指定位置	a(0,1)	a[0,1]
向量子集	a(1 to 4), a(1 until 5), a.slice(1,5)	a[1:5]
按照指定步长取子集	a(5 to 0 by -1)	a[5:0:-1]
指定开始位置至结尾	a(1 to -1)	a[1:]
最后一个元素	a(-1)	a[-1]
矩阵指定列	a(::, 2)	a[:,2]

```
scala> val a = DenseVector(1,2,3,4,5,6,7,8,9,10)
```

```
a: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> a(0)
res2: Int = 1
scala> a(1 to 4)
res4: breeze.linalg.DenseVector[Int] = DenseVector(2, 3, 4, 5)
scala> a(5 to 0 by -1)
res5: breeze.linalg.DenseVector[Int] = DenseVector(6, 5, 4, 3, 2, 1)
scala> a(1 to -1)
res6: breeze.linalg.DenseVector[Int] = DenseVector(2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> a( -1 )
res7: Int = 10
scala> val m = DenseMatrix((1.0,2.0,3.0), (3.0,4.0,5.0))
m: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
3.0 4.0 5.0
scala> m(0,1)
res8: Double = 2.0
scala> m(::,1)
res9: breeze.linalg.DenseVector[Double] = DenseVector(2.0, 4.0)
```

5.2.3 Breeze 元素操作

Breeze 元素操作如表 5-3 所示。

表 5-3 Breeze 元素操作

操作名称	Breeze 函数	对应 Numpy 函数
调整矩阵形状	a.reshape(3,2)	a.reshape(3,2)
矩阵转成向量	a.toDenseVector(Makes copy)	a.flatten()
复制下三角	lowerTriangular(a)	tril(a)
复制上三角	upperTriangular(a)	triu(a)
矩阵复制	a.copy	np.copy(a)
取对角线元素	diag(a)	diagonal(a)
子集赋数值	a(1 to 4) := 5.0	a[1:4]=5.0
子集赋向量	a(1 to 4) := DenseVector(1.0,2.0,3.0)	a[1:4]=[1.0 2.0 3.0]
矩阵赋值	a(1 to 3, 1 to 3) := 5.0	a[2:4, 2:4] = 5.0
矩阵列赋值	a(::, 2) := 5.0	a(:,3) = 5
垂直连接矩阵	DenseMatrix.vertcat(a,b)	[a;b]
横向连接矩阵	DenseMatrix.horzcat(a,b)	[a,b]
向量连接	DenseVector.vertcat(a,b)	[a b]

```
scala> val m = DenseMatrix((1.0,2.0,3.0), (3.0,4.0,5.0))
m: breeze.linalg.DenseMatrix[Double] =
```

```
1.0 2.0 3.0
3.0 4.0 5.0
scala> m.reshape(3, 2)
res11: breeze.linalg.DenseMatrix[Double] =
1.0 4.0
3.0 3.0
2.0 5.0
scala>m.toDenseVector
res12: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 3.0, 2.0, 4.0, 3.0,
5.0)
scala> val m = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0) , (7.0,8.0,9.0))
m: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
4.0 5.0 6.0
7.0 8.0 9.0
scala> val m = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0) , (7.0,8.0,9.0))
m: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
4.0 5.0 6.0
7.0 8.0 9.0
scala> lowerTriangular(m)
res19: breeze.linalg.DenseMatrix[Double] =
1.0 0.0 0.0
4.0 5.0 0.0
7.0 8.0 9.0
scala> upperTriangular(m)
res20: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
0.0 5.0 6.0
0.0 0.0 9.0
scala> m.copy
res21: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
4.0 5.0 6.0
7.0 8.0 9.0
scala> diag(m)
res22: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 5.0, 9.0)
scala> m(::, 2) := 5.0
res23: breeze.linalg.DenseVector[Double] = DenseVector(5.0, 5.0, 5.0)
```

```
scala> m
res24: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 5.0
4.0 5.0 5.0
7.0 8.0 5.0
scala> m(1 to 2,1 to 2) := 5.0
res32: breeze.linalg.DenseMatrix[Double] =
5.0 5.0
5.0 5.0
scala> m
res33: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 5.0
4.0 5.0 5.0
7.0 5.0 5.0
scala> val a = DenseVector(1,2,3,4,5,6,7,8,9,10)
a: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> a(1 to 4) := 5
res27: breeze.linalg.DenseVector[Int] = DenseVector(5, 5, 5, 5)
scala> a(1 to 4) := DenseVector(1,2,3,4)
res29: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4)
scala> a
res30: breeze.linalg.DenseVector[Int] = DenseVector(1, 1, 2, 3, 4, 6, 7, 8, 9,
10)
scala> val a1 = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0))
a1: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
4.0 5.0 6.0
scala> val a2 = DenseMatrix((1.0,1.0,1.0), (2.0,2.0,2.0))
a2: breeze.linalg.DenseMatrix[Double] =
1.0 1.0 1.0
2.0 2.0 2.0
scala> DenseMatrix.vertcat(a1,a2)
res34: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
4.0 5.0 6.0
1.0 1.0 1.0
2.0 2.0 2.0
scala> DenseMatrix.horzcat(a1,a2)
res35: breeze.linalg.DenseMatrix[Double] =
```

```
1.0 2.0 3.0 1.0 1.0 1.0
4.0 5.0 6.0 2.0 2.0 2.0
scala> val b1 = DenseVector(1,2,3,4)
b1: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4)
scala> val b2 = DenseVector(1,1,1,1)
b2: breeze.linalg.DenseVector[Int] = DenseVector(1, 1, 1, 1)
scala> DenseVector.vertcat(b1,b2)
res36: breeze.linalg.DenseVector[Int] = DenseVector(1, 2, 3, 4, 1, 1, 1, 1)
```

5.2.4　Breeze 数值计算函数

Breeze 数值计算函数如表 5-4 所示。

表 5-4　Breeze 数值计算函数

操作名称	Breeze 函数	对应 Numpy 函数
元素加法	a + b	a + b
元素乘法	a :* b	a * b
元素除法	a :/ b	a / b
元素比较	a :< b	a < b
元素相等	a :== b	a == b
元素追加	a :+= 1.0	a += 1
元素追乘	a :*= 2.0	a *= 2
向量点积	a dot b, a.t * b^T	dot(a,b)
元素最大值	max(a)	a.max()
元素最大值及位置	argmax(a)	a.argmax()

```
scala> val a = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0))
a: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
4.0 5.0 6.0
scala> val b = DenseMatrix((1.0,1.0,1.0), (2.0,2.0,2.0))
b: breeze.linalg.DenseMatrix[Double] =
1.0 1.0 1.0
2.0 2.0 2.0
scala> a + b
res37: breeze.linalg.DenseMatrix[Double] =
2.0 3.0 4.0
6.0 7.0 8.0
scala> a :* b
res38: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
8.0 10.0 12.0
```

```
scala> a :/ b
res39: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
2.0 2.5 3.0
scala> a :< b
res40: breeze.linalg.DenseMatrix[Boolean] =
false false false
false false false
scala> a :== b
res41: breeze.linalg.DenseMatrix[Boolean] =
true false false
false false false
scala> a :+= 1.0
res42: breeze.linalg.DenseMatrix[Double] =
2.0 3.0 4.0
5.0 6.0 7.0
scala> a :*= 2.0
res43: breeze.linalg.DenseMatrix[Double] =
4.0 6.0 8.0
10.0 12.0 14.0
scala> max(a)
res47: Double = 14.0
scala> argmax(a)
res48: (Int, Int) = (1,2)
scala> DenseVector(1, 2, 3, 4) dot DenseVector(1, 1, 1, 1)
res50: Int = 10
```

5.2.5 Breeze 求和函数

Breeze 求和函数如表 5-5 所示。

表 5-5 Breeze 求和函数

操作名称	Breeze 函数	对应 Numpy 函数
元素求和	sum(a)	a.sum()
每一列求和	sum(a, axis._0), sum(a(::,*))	sum(a,0)
每一行求和	sum(a,axis._1), sum(a(*, ::))	sum(a,1)
对角线元素和	trace(a)	a.trace()
累积和	accumulate(a)	a.cumsum()

```
scala> val a = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0) , (7.0,8.0,9.0))
a: breeze.linalg.DenseMatrix[Double] =
```

```
1.0 2.0 3.0
4.0 5.0 6.0
7.0 8.0 9.0
scala> sum(a)
res51: Double = 45.0
scala> sum(a, Axis._0)
res52: breeze.linalg.DenseMatrix[Double] = 12.0 15.0 18.0
scala> sum(a, Axis._1)
res53: breeze.linalg.DenseVector[Double] = DenseVector(6.0, 15.0, 24.0)
scala> trace(a)
res54: Double = 15.0
scala> accumulate(DenseVector(1, 2, 3, 4))
res56: breeze.linalg.DenseVector[Int] = DenseVector(1, 3, 6, 10)
```

5.2.6　Breeze 布尔函数

Breeze 布尔函数如表 5-6 所示。

表 5-6　Breeze 布尔函数

操作名称	Breeze 函数	对应 Numpy 函数
元素与操作	a :& b	a & b
元素或操作	a :| b	a | b
元素非操作	!a	~a
任意元素非零	any(a)	any(a)
所有元素非零	all(a)	all(a)

```
scala> val a = DenseVector(true, false, true)
a: breeze.linalg.DenseVector[Boolean] = DenseVector(true, false, true)
scala> val b = DenseVector(false, true, true)
b: breeze.linalg.DenseVector[Boolean] = DenseVector(false, true, true)
scala> a :& b
res57: breeze.linalg.DenseVector[Boolean] = DenseVector(false, false, true)
scala> a :| b
res58: breeze.linalg.DenseVector[Boolean] = DenseVector(true, true, true)
scala> !a
res59: breeze.linalg.DenseVector[Boolean] = DenseVector(false, true, false)
scala> val a = DenseVector(1.0, 0.0, -2.0)
a: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 0.0, -2.0)
scala> any(a)
res60: Boolean = true
```

```
scala> all(a)
res61: Boolean = false
```

5.2.7 Breeze 线性代数函数

Breeze 线性代数函数如表 5-7 所示。

表 5-7 Breeze 线性代数函数

操作名称	Breeze 函数	对应 Numpy 函数
线性求解	a \ b	linalg.solve(a,b)
转置	a.t	a.conj.transpose()
求行列式	det(a)	linalg.det(a)
求逆	inv(a)	linalg.inv(a)
求伪逆	pinv(a)	linalg.pinv(a)
求范数	norm(a)	norm(a)
特征值和特征向量	eigSym(a)	linalg.eig(a)[0]
特征值	val(er,ei,_) = eig(a)（实部与虚部分开）	lialg.eig(a)[0]
特征向量	eig(a)._3	
奇异值分解	val svd.SVD(u,s,v) = svd(a)	linalg.svd(a)
求矩阵的秩	rank(a)	rank(a)
矩阵长度	a.length	a.size
矩阵行数	a.rows	a.shape[0]
矩阵列数	a.cols	a.shape[1]

```
scala> val a = DenseMatrix((1.0,2.0,3.0), (4.0,5.0,6.0) , (7.0,8.0,9.0))
a: breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
4.0 5.0 6.0
7.0 8.0 9.0
scala> val b = DenseMatrix((1.0,1.0,1.0), (1.0,1.0,1.0) , (1.0,1.0,1.0))
b: breeze.linalg.DenseMatrix[Double] =
1.0 1.0 1.0
1.0 1.0 1.0
1.0 1.0 1.0
scala> a \ b
res74: breeze.linalg.DenseMatrix[Double] =
-2.5 -2.5 -2.5
4.0 4.0 4.0
-1.5 -1.5 -1.5
scala> a.t
res63: breeze.linalg.DenseMatrix[Double] =
```

```
1.0 4.0 7.0
2.0 5.0 8.0
3.0 6.0 9.0
scala> det(a)
res64: Double = 6.661338147750939E-16
scala> a \ b
res74: breeze.linalg.DenseMatrix[Double] =
-2.5 -2.5 -2.5
4.0 4.0 4.0
-1.5 -1.5 -1.5
scala> a.t
res63: breeze.linalg.DenseMatrix[Double] =
1.0 4.0 7.0
2.0 5.0 8.0
3.0 6.0 9.0
scala> det(a)
res64: Double = 6.661338147750939E-16
```

5.2.8　Breeze 取整函数

Breeze 取整函数如表 5-8 所示。

表 5-8　Breeze 取整函数

操作名称	Breeze 函数	对应 Numpy 函数
四舍五入	round(a)	around(a)
最小整数	ceil(a)	ceil(a)
最大整数	floor(a)	floor(a)
符号函数	signum(a)	sign(a)
取正数	abs(a)	abs(a)

```
scala> val a = DenseVector(1.2, 0.6, -2.3)
a: breeze.linalg.DenseVector[Double] = DenseVector(1.2, 0.6, -2.3)
scala> round(a)
res75: breeze.linalg.DenseVector[Long] = DenseVector(1, 1, -2)
scala> ceil(a)
res76: breeze.linalg.DenseVector[Double] = DenseVector(2.0, 1.0, -2.0)
scala> floor(a)
res77: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 0.0, -3.0)
scala> signum(a)
res78: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 1.0, -1.0)
```

```
scala> abs(a)
res79: breeze.linalg.DenseVector[Double] = DenseVector(1.2, 0.6, 2.3)
```

5.2.9　Breeze 三角函数

Breeze 三角函数包括：sin、sinh、asin、asinh cos、cosh、aco、acosh tan、tanh、atan、atanh atan2、sinc(x) 和 sincpi(x) 。

5.2.10　BLAS 向量运算

BLAS 按照功能被分为三个级别：

（1）Level 1：矢量-矢量运算，比如点积（ddot)、加法和数乘 （daxpy）、绝对值的和（dasum）等。

（2）Level 2：矩阵-矢量运算，最重要的函数是一般的矩阵向量乘法（dgemv）。

（3）Level 3：矩阵-矩阵运算，最重要的函数是一般的矩阵乘法（dgemm）。

每一种函数操作都区分不同数据类型（单精度、双精度、复数）。

1. BLAS 向量——向量运算

BLAS 向量——向量运算如表 5-9 所示。

表 5-9　BLAS 向量——向量运算

函数	说明
SROTG	Givens 旋转设置
SROTMG	改进 Givens 旋转设置
SROT	Givens 旋转
SROTM	改进 Givens 旋转
SSWAP	交换 x 和 y
SSCAL	常数 a 乘以向量 x()
SCOPY	把 x 复制到 y
SAXPY	向量 y+常数 a 乘以向量 x（y = a*x + y）
SDOT	点积
SDSDOT	扩展精度累积的点积
SNRM2	欧氏范数
SCNRM2	欧氏范数
SASUM	绝对值之和
ISAMAX	最大值位置

2. BLAS 矩阵-向量运算

（1）SGEMV 矩阵向量乘法。

（2）SGBMV 带状矩阵向量乘法。

（3）SSYMV 对称矩阵向量乘法。

（4）SSBMV 对称带状矩阵向量乘法。

（5）SSPMV 对称填充矩阵向量乘法。

（6）STRMV 三角矩阵向量乘法。
（7）STBMV 三角带状矩阵向量乘法。
（8）STPMV 三角填充矩阵向量乘法。
（9）STRSV 求解三角矩阵。
（10）STBSV 求解三角带状矩阵。
（11）STPSV 求解三角填充矩阵。
（12）SGER A := alpha*x*y’ + A。
（13）SSYR A := alpha*x*x’ + A。
（14）SSPR A := alpha*x*x’ + A。
（15）SSYR2 A := alpha*x*y’ + alpha*y*x’ + A。
（16）SSPR2 A := alpha*x*y’ + alpha*y*x’ + A。

3. BLAS 矩阵——矩阵运算函数脖子胳膊裤子

BLAS 矩阵运算函数如表 5-10 所示。

表 5-10　BLAS 矩阵——矩阵运算

函数	说明
SGEMM	矩阵乘法
SSYMM	对称矩阵乘法
SSYPK	对称矩阵的秩-k 修正
SSYR2K	对称矩阵的秩-2k 修正
STRMM	三角矩阵乘法
STRSM	多重右端的三角线性方程组求解

5.3 Spark MLlib 线性回归算法

5.3.1　线性回归算法理论基础

在统计学中，线性回归（Linear Regression）是利用称为线性回归方程的最小平方函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析。这种函数是一个或多个称为回归系数的模型参数的线性组合[69-73]。

回归分析中，只包括一个自变量和一个因变量，且二者的关系可用一条直线近似表示，这种回归分析称为一元线性回归分析。如果回归分析中包括两个或两个以上的自变量，且因变量和自变量之间是线性关系，则称为多元线性回归分析。

线性回归，对于初学者而言比较难理解，其实换个叫法（如线性拟合）可能就能理解线性回归是做什么的了，数学表达式如下：

$$h(x) = a_0 + a_1x_1 + a_2x_2 + \ldots a_nx_n + J(\theta) \tag{5-1}$$

其中 $h(x)$ 为预测函数， $a_i(i=1,2,..,n)x_i(i=1,2,..,n)$ 为估计参数，模型训练的目的就是计算出这些参数的值[74]。

而线性回归分析的整个过程可以简单描述为如下三个步骤：

（1）寻找合适的预测函数，即上文中的 $h(x)$，用来预测输入数据的判断结果。这个过程是非常关键的，需要对数据有一定的了解或分析，知道或者猜测预测函数的“大概”形式，比如是线性函数还是非线性函数，若是非线性的则无法用线性回归来得出高质量的结果。

（2）构造一个 Loss 函数（损失函数），该函数表示预测的输出（h）与训练数据标签之间的偏差，可以是二者之间的差（h-y）或者是其他的形式（如平方差开方）。综合考虑所有训练数据的“损失”，将 Loss 求和或者求平均，记为 $J(\theta)$ 函数，表示所有训练数据预测值与实际类别的偏差。

（3）显然，$J(\theta)$ 函数的值越小表示预测函数越准确（即 h 函数越准确），所以这一步需要做的是找到 $J(\theta)$ 函数的最小值。找函数的最小值有不同的方法，Spark 中采用的是梯度下降法（stochastic gradient descent，SGD)。

5.3.2 线性回归算法

无论是一元线性方程还是多元线性方程，可统一写成如下的格式：

$$h_\theta(x) = \theta^T X \tag{5-2}$$

求线性方程则演变成了求方程的参数 θ^T。

1. 梯度下降算法

为了得到目标线性方程，我们只需确定公式（5-2）中的 θ^T，同时为了确定所选定的 θ^T 效果好坏，通常情况下，使用一个损失函数(loss function)或者错误函数(error function)来评估 $h(x)$函数的好坏。该错误函数如公式(5-3)所示。前面乘上的 1/2 是为了在求导的时候，这个系数就不见了[75]。

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)^2 \tag{5-3}$$

2. 批量梯度下降算法

如前所述，求 θ^T 的问题演变成了求 $J(\theta)$ 的极小值问题，这里使用梯度下降法。而梯度下降法中的梯度方向由 $J(\theta)$ 对 θ 的偏导数确定，由于求的是极小值，因此梯度方向是偏导数的反方向[76]。

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \tag{5-4}$$

公式(5-4)中 α 为学习速率，当 α 过大时，有可能越过最小值；而 α 当过小时，容易造成迭代次数较多，收敛速度较慢。假如数据集中只有一条样本，所以公式(5-4)中

$$\begin{aligned}\frac{\partial}{\partial\theta_j}J(\theta)&=\frac{\partial}{\partial\theta_j}\frac{1}{2}\left(h_\theta(x)-y\right)^2\\&=2\cdot\frac{1}{2}\left(h_\theta(x)-y\right)\cdot\frac{\partial}{\partial\theta_j}\left(h_\theta(x)-y\right)\\&=\left(h_\theta(x)-y\right)\cdot\frac{\partial}{\partial\theta}\left(\sum_i^n\theta_i x_i-y\right)\\&=\left(h_\theta(x)-y\right)x_j\end{aligned}\tag{5-5}$$

就演变成：

$$\theta_j:=\theta_j+\alpha\left(y^{(i)}-h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}\tag{5-6}$$

当样本数量 m 不为 1 时，将公式(5-4)中 $\frac{\partial}{\partial\theta_j}J(\theta)$ 由公式(5-3)带入求偏导，那么每个参数沿梯度方向的变化值由公式(5-7)求得[77]。

$$\theta_j:=\theta_j+\alpha\sum_{i=1}^m\left(y^{(i)}-h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}\tag{5-7}$$

初始时 θ^T 可设为 0，然后迭代使用公式(5-7)计算 θ^T 中的每个参数，直至收敛为止。由于每次迭代计算 θ^T 时，都使用了整个样本集，因此我们称该梯度下降算法为批量梯度下降算法(batch gradient descent)[78]。

3. 随机梯度下降算法

当样本集数据量 m 很大时，批量梯度下降算法每迭代一次的复杂度为 O(mn)，复杂度很高。因此，为了减少复杂度，当 m 很大时，更多时候使用随机梯度下降算法(stochastic gradient descent)，算法如下所示：

$$\begin{aligned}&loop\{\\&\quad for\, i=1\quad to\quad m\quad\{\\&\qquad\theta_j:=\theta_j+\alpha\left(y^{(i)}-h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}\qquad(for\quad every\quad j)\\&\quad\}\\&\}\end{aligned}\tag{5-8}$$

即每读取一条样本，就迭代对 θ^T 进行更新，然后判断其是否收敛，若没收敛，则继续读取样本进行处理，如果所有样本都读取完毕了，则循环重新从头开始读取样本进行处理。

这样迭代一次的算法复杂度为 O(n)。对于大数据集，很有可能只需读取一小部分数据，函数 $J(\theta)$就收敛了。比如样本集数据量为 100 万，有可能读取几千条或几万条时，函数就达到了收敛值。所以当数据量很大时，更倾向于选择随机梯度下降算法。

4. 最小二乘法

将训练特征表示为 X 矩阵，结果表示成 y 向量，仍然是线性回归模型，误差函数不变。那么 θ 可以直接由下面公式得出

$$\theta = \left(X^T X\right)^{-1} X^T \vec{y} \tag{5-9}$$

5.3.3 Spark MLlib Linear Regression 源码分析

1. 线性回归算法的 train 方法

线性回归算法的 train 方法，由 LinearRegressionWithSGD 类的 object 定义了 train 函数。

```
package org.apache.Spark.mllib.regression
  def train(
      input: RDD[LabeledPoint],
      numIterations: Int,
      stepSize: Double,
      miniBatchFraction: Double): LinearRegressionModel = {
    new LinearRegressionWithSGD(stepSize, numIterations,
miniBatchFraction).run(input)
  }
```

Input 为输入样本，numIterations 为迭代次数，stepSize 为步长，miniBatchFraction 为迭代因子。创建一个 LinearRegressionWithSGD 对象，初始化梯度下降算法。Run 方法来自于继承父类 GeneralizedLinearAlgorithm，实现方法如下。

2. LinearRegressionWithSGD 中 run 方法的实现

```
package org.apache.spark.mllib.regression
/**
   * Run the algorithm with the configured parameters on an input RDD
   * of LabeledPoint entries starting from the initial weights provided.
   */
  def run(input: RDD[LabeledPoint], initialWeights: Vector): M = {
//特征维度赋值。
    if (numFeatures < 0) {
      numFeatures = input.map(_.features.size).first()
    }
//输入样本数据检测。
    if (input.getStorageLevel == StorageLevel.NONE) {
      logWarning("The input data is not directly cached, which may hurt
```

```
performance if its"
        + " parent RDDs are also uncached.")
    }
//输入样本数据检测。
    // Check the data properties before running the optimizer
    if (validateData && !validators.forall(func => func(input))) {
      thrownew SparkException("Input validation failed.")
    }
val scaler = if (useFeatureScaling) {
      new StandardScaler(withStd = true, withMean
= false).fit(input.map(_.features))
    } else {
      null
    }
//输入样本数据处理，输出 data(label, features)格式。
//addIntercept：是否增加θ0常数项，若增加，则增加 x0=1项。
    // Prepend an extra variable consisting of all 1.0's for the intercept.
    // TODO: Apply feature scaling to the weight vector instead of input data.
    val data =
      if (addIntercept) {
        if (useFeatureScaling) {
          input.map(lp => (lp.label,
appendBias(scaler.transform(lp.features)))).cache()
        } else {
          input.map(lp => (lp.label, appendBias(lp.features))).cache()
        }
      } else {
        if (useFeatureScaling) {
          input.map(lp => (lp.label, scaler.transform(lp.features))).cache()
        } else {
          input.map(lp => (lp.label, lp.features))
        }
      }
//初始化权重。
// addIntercept：是否增加θ0常数项，若增加，则权重增加θ0。
    /**
     * TODO: For better convergence, in logistic regression, the intercepts
should be computed
     * from the prior probability distribution of the outcomes; for linear
```

```
regression,
     * the intercept should be set as the average of response.
     */
    val initialWeightsWithIntercept = if (addIntercept && numOfLinearPredictor
== 1) {
      appendBias(initialWeights)
    } else {
      /** If `numOfLinearPredictor > 1`, initialWeights already contains
intercepts. */
      initialWeights
    }
//权重优化，进行梯度下降学习，返回最优权重。
    val weightsWithIntercept = optimizer.optimize(data,
initialWeightsWithIntercept)

    val intercept = if (addIntercept && numOfLinearPredictor == 1) {
      weightsWithIntercept(weightsWithIntercept.size - 1)
    } else {
      0.0
    }

    var weights = if (addIntercept && numOfLinearPredictor == 1) {
      Vectors.dense(weightsWithIntercept.toArray.slice(0,
weightsWithIntercept.size - 1))
    } else {
      weightsWithIntercept
    }

    createModel(weights, intercept)
}
```

其中 optimizer.optimize(data, initialWeightsWithIntercept) 是线性回归实现的核心。oprimizer 的类型为 GradientDescent，optimize 方法中主要调用 GradientDescent 伴生对象的 runMiniBatchSGD 方法，返回当前迭代产生的最优特征权重向量。

GradientDescentd 对象中 optimize 实现方法如下。

3. optimize 实现方法

```
package org.apache.spark.mllib.optimization
/**
```

```
   * :: DeveloperApi ::
   * Runs gradient descent on the given training data.
   * @param data training data
   * @param initialWeights initial weights
   * @return solution vector
   */
  @DeveloperApi
  def optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector =
{
    val (weights, _) = GradientDescent.runMiniBatchSGD(
      data,
      gradient,
      updater,
      stepSize,
      numIterations,
      regParam,
      miniBatchFraction,
      initialWeights)
    weights
  }

}
```

在 optimize 方法中，调用了 GradientDescent.runMiniBatchSGD 方法，其 runMiniBatchSGD 实现方法如下：

```
/**
   * Run stochastic gradient descent (SGD) in parallel using mini batches.
   * In each iteration, we sample a subset (fraction miniBatchFraction) of the
total data
   * in order to compute a gradient estimate.
   * Sampling, and averaging the subgradients over this subset is performed
using one standard
   * spark map-reduce in each iteration.
   *
   * @param data - Input data for SGD. RDD of the set of data examples, each
of
   *                   the form (label, [feature values]).
   * @param gradient - Gradient object (used to compute the gradient of the
loss function of
```

```
   *                   one single data example)
   * @param updater - Updater function to actually perform a gradient step in
a given direction.
   * @param stepSize - initial step size for the first step
   * @param numIterations - number of iterations that SGD should be run.
   * @param regParam - regularization parameter
   * @param miniBatchFraction - fraction of the input data set that should be
used for
   *                            one iteration of SGD. Default value 1.0.
   *
   * @return A tuple containing two elements. The first element is a column
matrix containing
   *         weights for every feature, and the second element is an array
containing the
   *         stochastic loss computed for every iteration.
   */
  def runMiniBatchSGD(
      data: RDD[(Double, Vector)],
      gradient: Gradient,
      updater: Updater,
      stepSize: Double,
      numIterations: Int,
      regParam: Double,
      miniBatchFraction: Double,
      initialWeights: Vector): (Vector, Array[Double]) = {
//历史迭代误差数组
    val stochasticLossHistory = new ArrayBuffer[Double](numIterations)
//样本数据检测，若为空，返回初始值。
    val numExamples = data.count()

    // if no data, return initial weights to avoid NaNs
    if (numExamples == 0) {
      logWarning("GradientDescent.runMiniBatchSGD returning initial weights,
no data found")
      return (initialWeights, stochasticLossHistory.toArray)
    }
// miniBatchFraction 值检测。
    if (numExamples * miniBatchFraction < 1) {
      logWarning("The miniBatchFraction is too small")
```

```
    }
// weights 权重初始化。
    // Initialize weights as a column vector
    var weights = Vectors.dense(initialWeights.toArray)
    val n = weights.size

    /**
     * For the first iteration, the regVal will be initialized as sum of
weight squares
     * if it's L2 updater; for L1 updater, the same logic is followed.
     */
    var regVal = updater.compute(
      weights, Vectors.dense(new Array[Double](weights.size)), 0, 1,
regParam)._2
// weights 权重迭代计算。
    for (i <- 1 to numIterations) {
      val bcWeights = data.context.broadcast(weights)
      // Sample a subset (fraction miniBatchFraction) of the total data
      // compute and sum up the subgradients on this subset (this is one map-
reduce)
//采用 treeAggregate 的 RDD 方法，进行聚合计算，计算每个样本的权重向量、误差值，然后对所有样
本权重向量及误差值进行累加。
      val (gradientSum, lossSum, miniBatchSize) = data.sample(false,
miniBatchFraction, 42 + i)
        .treeAggregate((BDV.zeros[Double](n), 0.0, 0L))(
          seqOp = (c, v) => {
            // c: (grad, loss, count), v: (label, features)
            val l = gradient.compute(v._2, v._1, bcWeights.value,
Vectors.fromBreeze(c._1))
            (c._1, c._2 + l, c._3 + 1)
          },
          combOp = (c1, c2) => {
            // c: (grad, loss, count)
            (c1._1 += c2._1, c1._2 + c2._2, c1._3 + c2._3)
          })
//保存本次迭代误差值，以及更新 weights 权重向量。
      if (miniBatchSize > 0) {
        /**
         * NOTE(Xinghao): lossSum is computed using the weights from the
```

```
previous iteration
         * and regVal is the regularization value computed in the previous
iteration as well.
         */
        stochasticLossHistory.append(lossSum / miniBatchSize + regVal)
        val update = updater.compute(
          weights, Vectors.fromBreeze(gradientSum / miniBatchSize.toDouble),
stepSize, i, regParam)
        weights = update._1
        regVal = update._2
      } else {
        logWarning(s"Iteration ($i/$numIterations). The size of sampled batch
is zero")
      }
    }

    logInfo("GradientDescent.runMiniBatchSGD finished. Last 10 stochastic
losses %s".format(
      stochasticLossHistory.takeRight(10).mkString(", ")))

    (weights, stochasticLossHistory.toArray)

  }
```

runMiniBatchSGD 的输入、输出参数说明如下：

- Data：样本输入数据，格式 (label, [feature values])。
- gradient：梯度对象，用于对每个样本计算梯度及误差。
- updater：权重更新对象，用于每次更新权重。
- stepSize：初始步长。
- numIterations：迭代次数。
- regParam：正则化参数。
- miniBatchFraction：迭代因子。
- 返回结果(Vector, Array[Double])：第一个为权重，第二个为每次迭代的误差值。

在 MiniBatchSGD 中主要实现对输入数据集进行迭代抽样，通过使用 LeastSquaresGradient 作为梯度下降算法，使用 SimpleUpdater 作为更新算法，不断对抽样数据集进行迭代计算从而找出最优的特征权重向量解。在 LinearRegressionWithSGD 中定义如下：

```
privateval gradient = new LeastSquaresGradient()
  privateval updater = new SimpleUpdater()
```

```
  overrideval optimizer = new GradientDescent(gradient, updater)
    .setStepSize(stepSize)
    .setNumIterations(numIterations)
.setMiniBatchFraction(miniBatchFraction)
```

4. gradient & updater

（1）gradient

```
/**
 * :: DeveloperApi ::
 * Compute gradient and loss for a Least-squared loss function, as used in
linear regression.
 * This is correct for the averaged least squares loss function (mean squared
error)
 *              L = 1/2n ||A weights-y||^2
 * See also the documentation for the precise formulation.
 */
@DeveloperApi
class LeastSquaresGradient extends Gradient {
//计算当前计算对象的类标签与实际类标签值之差
//计算当前平方梯度下降值
//计算权重的更新值
//返回当前训练对象的特征权重向量和误差
  overridedef compute(data: Vector, label: Double, weights: Vector): (Vector,
Double) = {
    val diff = dot(data, weights) - label
    val loss = diff * diff / 2.0
    val gradient = data.copy
    scal(diff, gradient)
    (gradient, loss)
  }

  overridedef compute(
      data: Vector,
      label: Double,
      weights: Vector,
      cumGradient: Vector): Double = {
    val diff = dot(data, weights) - label
    axpy(diff, data, cumGradient)
```

```
    diff * diff / 2.0
  }
}
```

（2）updater

```
/**
 * :: DeveloperApi ::
 * A simple updater for gradient descent *without* any regularization.
 * Uses a step-size decreasing with the square root of the number of
iterations.
 */
//weihtsOld:上一次迭代计算后的特征权重向量
//gradient:本次迭代计算的特征权重向量
//stepSize:迭代步长
//iter:当前迭代次数
//regParam:回归参数
//以当前迭代次数的平方根的倒数作为本次迭代趋近(下降)的因子
//返回本次剃度下降后更新的特征权重向量
@DeveloperApi
class SimpleUpdater extends Updater {
  overridedef compute(
      weightsOld: Vector,
      gradient: Vector,
      stepSize: Double,
      iter: Int,
      regParam: Double): (Vector, Double) = {
    val thisIterStepSize = stepSize / math.sqrt(iter)
    val brzWeights: BV[Double] = weightsOld.toBreeze.toDenseVector
    brzAxpy(-thisIterStepSize, gradient.toBreeze, brzWeights)

    (Vectors.fromBreeze(brzWeights), 0)
  }
}
```

5. MLlib Linear Regression 实例

```
//1 读取样本数据
    valdata_path = "/user/tmp/lpsa.data"
    valdata = sc.textFile(data_path)
    valexamples = data.map { line =>
```

```
      valparts = line.split(',')
      LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split('
').map(_.toDouble)))
    }.cache()

//2 样本数据划分训练样本与测试样本
    valsplits = examples.randomSplit(Array(0.8, 0.2))
    valtraining = splits(0).cache()
    valtest = splits(1).cache()
    valnumTraining = training.count()
    valnumTest = test.count()
    println(s"Training: $numTraining, test: $numTest.")

//3 新建线性回归模型，并设置训练参数
    valnumIterations = 100
    valstepSize = 1
    valminiBatchFraction = 1.0
    valmodel =
LinearRegressionWithSGD.train(training, numIterations, stepSize, miniBatchFrac
tion)
//4 对测试样本进行测试
    valprediction = model.predict(test.map(_.features))
    valpredictionAndLabel = prediction.zip(test.map(_.label))
//5计算测试误差
    valloss = predictionAndLabel.map {
      case (p, l) =>
        valerr = p - l
        err * err
    }.reduce(_ + _)
    valrmse = math.sqrt(loss / numTest)
    println(s"Test RMSE = $rmse.")
```

5.4 Spark MLlib 逻辑回归算法

逻辑回归作为分类算法的一种，在互联网领域中的预测、判别中应用得非常广泛，像广告投放中的点击率预估、推荐算法中的模型融合等。本节简要介绍逻辑回归的算法，以及在

Spark MLlib 中的实现解析[79]。

5.4.1 逻辑回归算法

逻辑回归其实是一个分类问题，此类问题的模型训练，基本上分为 3 步骤：

（1）第一步要寻找假设预测函数 h，构造的假设函数为：

$$h_\theta(x)=g\left(\theta^T x\right)=\frac{1}{1+e^{-\theta^T x}} \tag{5-10}$$

Logistic 函数（或称为 Sigmoid 函数），函数形式为：

$$g(z)=\frac{1}{1+e^{-z}} \tag{5-11}$$

在线性回归的函数基础上，加上一个 Sigmoid 函数进行 Norm，把函数值输出在 0 到 1 的范围内，函数的值有特殊的含义，它表示结果取 1 的概率，因此对于输入 x 分类结果为类别 1 和类别 0 的概率分别为：

$$\begin{aligned}&P(y=1 \mid x;\theta)=h_\theta(x)\\&P(y=0 \mid x;\theta)=1-h_\theta(x)\end{aligned} \tag{5-12}$$

（2）第二步要构造损失函数 J，基于最大似然估计推导出：

$$J(\theta)=-\frac{1}{m}\left[\ \sum_{1}^{m} y^{(i)}\log h_\theta\left(x^{(i)}\right)+\left(1-y^{(i)}\right)\log\left(1-h_\theta x^{(i)}\right)\ \right] \tag{5-13}$$

其中：$h_\theta(x)=1+\dfrac{1}{1+e^{-\theta^T x}}$。

（3）第三步求得使 $J(\theta)$最小值时的参数 θ，解决这个问题的做法是随机给定一个初始值 θ，通过迭代，在每次迭代中计算损失函数 $J(\theta)$的下降方向并更新 θ，直到目标函数收敛稳定在最小点。

图 5-5 所示的迭代优化算法就是损失函数 $J(\theta)$的下降方向的计算[80]。

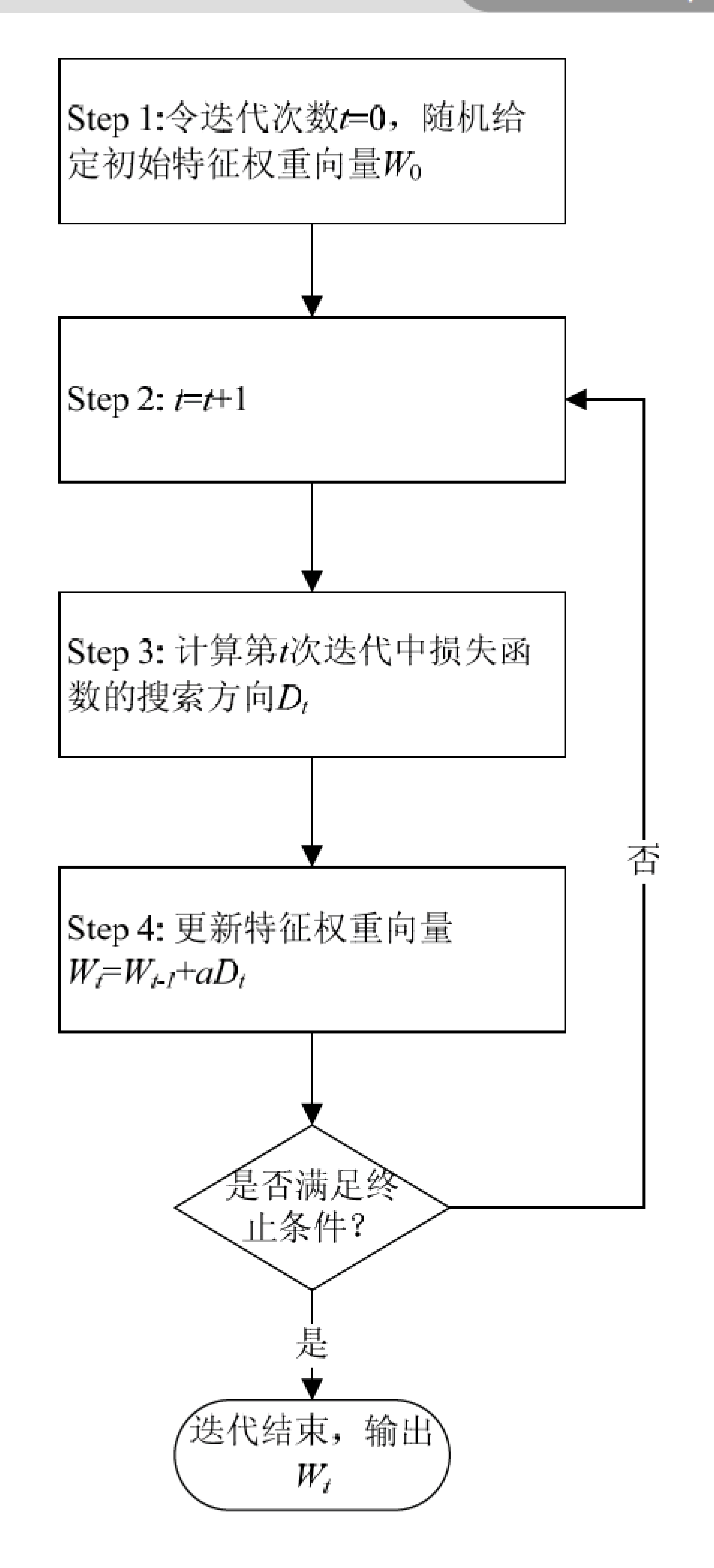


图 5-5 迭代优化算法计算损失函数 $J(\theta)$的下降方向

在实际应用过程中，为了增强模型的泛化能力，防止我们训练的模型过拟合，特别是对于大量的稀疏特征，模型复杂度比较高，需要进行降维，我们需要保证在训练误差最小化的基础上，通过加上正则化项减小模型复杂度。在逻辑回归中，有 L_1、L_2进行正则化[81]。

损失函数如下：

$$J(\theta)=\frac{1}{2m}\sum_{i=1}^{n}\left(h_\theta(x_i)-y_i\right)^2+\lambda\sum_{j=1}^{n}\theta_j^2 \qquad (5\text{-}14)$$

在损失函数里加入一个正则化项，正则化项就是权重的 L_1 或者 L_2 范数乘以一个 λ，用来控制损失函数和正则化项的比重。直观地理解，首先防止过拟合的目的就是防止最后训练出来的模型过分地依赖某一个特征，当最小化损失函数的时候，某一维度很大，拟合出来的函数值与真实的值之间的差距很小，通过正则化可以使整体的消耗变大，从而避免了过分依赖某一维度的结果。当然加正则化的前提是特征值要进行归一化[82]。

L_2 正则化假设模型参数服从高斯分布，L_2 正则化函数比 L_1 更光滑，所以更容易计算；L_1

假设模型参数服从拉普拉斯分布，L_1 正则化具备产生稀疏解的功能，从而具备特征的能力。

L_1 的计算公式：

$$shrinkageVal = regParam * \frac{stepSize}{\sqrt{iter}}$$

$$weight := signum(weight - \frac{stepSize}{\sqrt{iter}} gradient) * \max(0, 0, abs(weight - \frac{stepSize}{\sqrt{iter}} gradient) - shrinkageVal) \tag{5-15}$$

使用了 L_1 regularization（R(w) = ||w||），利用 soft-thresholding 方法求解，参数 weight 更新规则参照 signum 符号函数，它的取值如下：

$$signum(x) = \begin{cases} x > 0 & 1 \\ x = 0 & 0 \\ x < 0 & -1 \end{cases} \tag{5-16}$$

使用了 L_2 regularization（R(w) = 1/2 ||w||^2），参数 weights 更新规则为：

$$weight := weight - \frac{stepSize}{\sqrt{iter}} * (gradient + regParam * weight) \tag{5-17}$$

5.4.2 Spark MLlib Logistic Regression 源码分析

1. LogisticRegressionWithSGD

Logistic 回归算法的 train 方法，由 LogisticRegressionWithSGD 类的 object 定义了 train 函数，在 train 函数中新建了 LogisticRegressionWithSGD 对象。

```
package org.apache.Spark.mllib.classification
//1 类：LogisticRegressionWithSGD
class LogisticRegressionWithSGD private[mllib] (
    privatevar stepSize: Double,
    privatevar numIterations: Int,
    privatevar regParam: Double,
    privatevar miniBatchFraction: Double)
  extends GeneralizedLinearAlgorithm[LogisticRegressionModel] with Serializabl
e {

  privateval gradient = new LogisticGradient()
  privateval updater = new SquaredL2Updater()
  overrideval optimizer = new GradientDescent(gradient, updater)
    .setStepSize(stepSize)
```

```
    .setNumIterations(numIterations)
    .setRegParam(regParam)
    .setMiniBatchFraction(miniBatchFraction)
  overrideprotectedval validators = List(DataValidators.binaryLabelValidator)

  /**
   * Construct a LogisticRegression object with default parameters: {stepSize:
1.0,
   * numIterations: 100, regParm: 0.01, miniBatchFraction: 1.0}.
   */
  defthis() = this(1.0, 100, 0.01, 1.0)

  overrideprotected[mllib] def createModel(weights: Vector, intercept: Double)
= {
    new LogisticRegressionModel(weights, intercept)
 }
```

LogisticRegressionWithSGD 类中参数说明：

- stepSize：迭代步长，默认为 1.0。
- numIterations：迭代次数，默认为 100。
- regParam：正则化参数，默认值为 0.0。
- miniBatchFraction：每次迭代参与计算的样本比例，默认为 1.0。
- gradient：LogisticGradient()，Logistic 梯度下降。
- updater：SquaredL2Updater()，正则化，L2 范数。
- optimizer：GradientDescent(gradient, updater)，梯度下降最优化计算。

```
// 2 train 方法
object LogisticRegressionWithSGD {
  /**
   * Train a logistic regression model given an RDD of (label, features) pairs.
We run a fixed
   * number of iterations of gradient descent using the specified step size.
Each iteration uses
   * `miniBatchFraction` fraction of the data to calculate the gradient. The
weights used in
   * gradient descent are initialized using the initial weights provided.
   * NOTE: Labels used in Logistic Regression should be {0, 1}
   *
   * @param input RDD of (label, array of features) pairs.
```

```
   * @param numIterations Number of iterations of gradient descent to run.
   * @param stepSize Step size to be used for each iteration of gradient
descent.
   * @param miniBatchFraction Fraction of data to be used per iteration.
   * @param initialWeights Initial set of weights to be used. Array should be
equal in size to
   *        the number of features in the data.
   */
  def train(
      input: RDD[LabeledPoint],
      numIterations: Int,
      stepSize: Double,
      miniBatchFraction: Double,
      initialWeights: Vector): LogisticRegressionModel = {
    new LogisticRegressionWithSGD(stepSize, numIterations, 0.0,
miniBatchFraction)
      .run(input, initialWeights)
  }
}
```

train 参数说明：

- input：样本数据，分类标签 lable 只能是 1.0 和 0.0 两种，feature 为 double 类型。
- numIterations：迭代次数，默认为 100。
- stepSize：迭代步长，默认为 1.0。
- miniBatchFraction：每次迭代参与计算的样本比例，默认为 1.0。
- initialWeights：初始权重，默认为 0 向量。

2. LogisticRegressionWithSGD 中 run 方法的实现

run 方法来自于继承父类 GeneralizedLinearAlgorithm，实现方法如下。

```
package org.apache.spark.mllib.regression
/**
   * Run the algorithm with the configured parameters on an input RDD
   * of LabeledPoint entries starting from the initial weights provided.
   */
  def run(input: RDD[LabeledPoint], initialWeights: Vector): M = {
//特征维度赋值。
    if (numFeatures < 0) {
      numFeatures = input.map(_.features.size).first()
```

```
    }
//输入样本数据检测。
    if (input.getStorageLevel == StorageLevel.NONE) {
      logWarning("The input data is not directly cached, which may hurt
performance if its"
        + " parent RDDs are also uncached.")
    }
//输入样本数据检测。
    // Check the data properties before running the optimizer
    if (validateData && !validators.forall(func => func(input))) {
      thrownew SparkException("Input validation failed.")
    }
val scaler = if (useFeatureScaling) {
      new StandardScaler(withStd = true, withMean
= false).fit(input.map(_.features))
    } else {
      null
    }
// 输入样本数据处理，输出 data(label, features)格式。
// addIntercept：是否增加 θ0常数项，若增加，则增加 x0=1项。
    // Prepend an extra variable consisting of all 1.0's for the intercept.
    // TODO: Apply feature scaling to the weight vector instead of input data.
    val data =
      if (addIntercept) {
        if (useFeatureScaling) {
          input.map(lp => (lp.label,
appendBias(scaler.transform(lp.features)))).cache()
        } else {
          input.map(lp => (lp.label, appendBias(lp.features))).cache()
        }
      } else {
        if (useFeatureScaling) {
          input.map(lp => (lp.label, scaler.transform(lp.features))).cache()
        } else {
          input.map(lp => (lp.label, lp.features))
        }
      }
//初始化权重。
//addIntercept：是否增加 θ0常数项，若增加，则权重增加 θ0。
```

```
    /**
     * TODO: For better convergence, in logistic regression, the intercepts
should be computed
     * from the prior probability distribution of the outcomes; for linear
regression,
     * the intercept should be set as the average of response.
     */
    val initialWeightsWithIntercept = if (addIntercept && numOfLinearPredictor
== 1) {
      appendBias(initialWeights)
    } else {
      /** If `numOfLinearPredictor > 1`, initialWeights already contains
intercepts. */
      initialWeights
    }
//权重优化，进行梯度下降学习，返回最优权重。
    val weightsWithIntercept = optimizer.optimize(data,
initialWeightsWithIntercept)

    val intercept = if (addIntercept && numOfLinearPredictor == 1) {
      weightsWithIntercept(weightsWithIntercept.size - 1)
    } else {
      0.0
    }

    var weights = if (addIntercept && numOfLinearPredictor == 1) {
      Vectors.dense(weightsWithIntercept.toArray.slice(0,
weightsWithIntercept.size - 1))
    } else {
      weightsWithIntercept
    }

    createModel(weights, intercept)
}
```

其中 optimizer.optimize(data, initialWeightsWithIntercept) 是逻辑回归实现的核心。oprimizer 的类型为 GradientDescent，optimize 方法调用 GradientDescent 对象的 runMiniBatchSGD 方法，返回当前迭代产生的最优特征权重向量。

3. GradientDescentd 对象中 optimize 实现方法

```
package org.apache.spark.mllib.optimization
/**
   * :: DeveloperApi ::
   * Runs gradient descent on the given training data.
   * @param data training data
   * @param initialWeights initial weights
   * @return solution vector
   */
  @DeveloperApi
  def optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector =
{
    val (weights, _) = GradientDescent.runMiniBatchSGD(
      data,
      gradient,
      updater,
      stepSize,
      numIterations,
      regParam,
      miniBatchFraction,
      initialWeights)
    weights
  }

}
```

在 optimize 方法中，调用了 GradientDescent.runMiniBatchSGD 方法，其 runMiniBatchSGD 实现方法如下：

```
/**
   * Run stochastic gradient descent (SGD) in parallel using mini batches.
   * In each iteration, we sample a subset (fraction miniBatchFraction) of the
total data
   * in order to compute a gradient estimate.
   * Sampling, and averaging the subgradients over this subset is performed
using one standard
   * spark map-reduce in each iteration.
   *
   * @param data - Input data for SGD. RDD of the set of data examples, each
```

```
of
   *               the form (label, [feature values]).
   * @param gradient - Gradient object (used to compute the gradient of the
loss function of
   *                   one single data example)
   * @param updater - Updater function to actually perform a gradient step in
a given direction.
   * @param stepSize - initial step size for the first step
   * @param numIterations - number of iterations that SGD should be run.
   * @param regParam - regularization parameter
   * @param miniBatchFraction - fraction of the input data set that should be
used for
   *                            one iteration of SGD. Default value 1.0.
   *
   * @return A tuple containing two elements. The first element is a column
matrix containing
   *          weights for every feature, and the second element is an array
containing the
   *          stochastic loss computed for every iteration.
   */
  def runMiniBatchSGD(
      data: RDD[(Double, Vector)],
      gradient: Gradient,
      updater: Updater,
      stepSize: Double,
      numIterations: Int,
      regParam: Double,
      miniBatchFraction: Double,
      initialWeights: Vector): (Vector, Array[Double]) = {
//历史迭代误差数组
    val stochasticLossHistory = new ArrayBuffer[Double](numIterations)
//样本数据检测，若为空，返回初始值。
    val numExamples = data.count()

    // if no data, return initial weights to avoid NaNs
    if (numExamples == 0) {
      logWarning("GradientDescent.runMiniBatchSGD returning initial weights,
no data found")
      return (initialWeights, stochasticLossHistory.toArray)
```

```
    }
//miniBatchFraction 值检测。
    if (numExamples * miniBatchFraction < 1) {
      logWarning("The miniBatchFraction is too small")
    }
//weights 权重初始化。
    // Initialize weights as a column vector
    var weights = Vectors.dense(initialWeights.toArray)
    val n = weights.size

    /**
     * For the first iteration, the regVal will be initialized as sum of
weight squares
     * if it's L2 updater; for L1 updater, the same logic is followed.
     */
    var regVal = updater.compute(
      weights, Vectors.dense(new Array[Double](weights.size)), 0, 1,
regParam)._2
// weights 权重迭代计算。
    for (i <- 1 to numIterations) {
      val bcWeights = data.context.broadcast(weights)
      // Sample a subset (fraction miniBatchFraction) of the total data
      // compute and sum up the subgradients on this subset (this is one map-
reduce)
//采用 treeAggregate 的 RDD 方法，进行聚合计算，计算每个样本的权重向量、误差值，然后对所有样
本权重向量及误差值进行累加。
//sample 是根据 miniBatchFraction 指定的比例随机采样相应数量的样本 。
      val (gradientSum, lossSum, miniBatchSize) = data.sample(false,
miniBatchFraction, 42 + i)
        .treeAggregate((BDV.zeros[Double](n), 0.0, 0L))(
          seqOp = (c, v) => {
            // c: (grad, loss, count), v: (label, features)
            val l = gradient.compute(v._2, v._1, bcWeights.value,
Vectors.fromBreeze(c._1))
            (c._1, c._2 + l, c._3 + 1)
          },
          combOp = (c1, c2) => {
            // c: (grad, loss, count)
            (c1._1 += c2._1, c1._2 + c2._2, c1._3 + c2._3)
```

```
          })
//保存本次迭代误差值，以及更新 weights 权重向量。
      if (miniBatchSize > 0) {
        /**
         * NOTE(Xinghao): lossSum is computed using the weights from the
previous iteration
         * and regVal is the regularization value computed in the previous
iteration as well.
         */
//updater.compute 更新 weights 矩阵和 regVal（正则化项）。根据本轮迭代中的 gradient 和
loss 的变化以及正则化项计算更新之后的 weights 和 regVal。
        stochasticLossHistory.append(lossSum / miniBatchSize + regVal)
        val update = updater.compute(
          weights, Vectors.fromBreeze(gradientSum / miniBatchSize.toDouble),
stepSize, i, regParam)
        weights = update._1
        regVal = update._2
      } else {
        logWarning(s"Iteration ($i/$numIterations). The size of sampled batch
is zero")
      }
    }

    logInfo("GradientDescent.runMiniBatchSGD finished. Last 10 stochastic
losses %s".format(
      stochasticLossHistory.takeRight(10).mkString(", ")))

    (weights, stochasticLossHistory.toArray)

  }
```

runMiniBatchSGD 的输入、输出参数说明：

- data：样本输入数据，格式 (label, [feature values])。
- gradient：梯度对象，用于对每个样本计算梯度及误差。
- updater：权重更新对象，用于每次更新权重。
- stepSize：初始步长。
- numIterations：迭代次数。
- regParam：正则化参数。
- miniBatchFraction：迭代因子，每次迭代参与计算的样本比例。

返回结果(Vector, Array[Double])，第一个为权重，每二个为每次迭代的误差值。

在 MiniBatchSGD 中主要实现对输入数据集进行迭代抽样，通过使用 LogisticGradient 作为梯度下降算法，使用 SquaredL2Updater 作为更新算法，不断对抽样数据集进行迭代计算从而找出最优的特征权重向量解。在 LinearRegressionWithSGD 中定义如下：

```
privateval gradient = new LogisticGradient()
privateval updater = new SquaredL2Updater()
overrideval optimizer = new GradientDescent(gradient, updater)
  .setStepSize(stepSize)
  .setNumIterations(numIterations)
  .setRegParam(regParam)
  .setMiniBatchFraction(miniBatchFraction)
```

4. runMiniBatchSGD 方法中调用了 gradient.compute、updater.compute 两个方法

（1）gradient

```
//计算当前计算对象的类标签与实际类标签值之差
//计算当前平方梯度下降值
//计算权重的更新值
//返回当前训练对象的特征权重向量和误差
class LogisticGradient(numClasses: Int) extends Gradient {

  defthis() = this(2)

  overridedef compute(data: Vector, label: Double, weights: Vector): (Vector,
Double) = {
    val gradient = Vectors.zeros(weights.size)
    val loss = compute(data, label, weights, gradient)
    (gradient, loss)
  }

  overridedef compute(
      data: Vector,
      label: Double,
      weights: Vector,
      cumGradient: Vector): Double = {
    val dataSize = data.size

    // (weights.size / dataSize + 1) is number of classes
    require(weights.size % dataSize == 0 && numClasses == weights.size /
```

```
dataSize + 1)
    numClasses match {
      case2 =>
        /**
         * For Binary Logistic Regression.
         *
         * Although the loss and gradient calculation for multinomial one is
more generalized,
         * and multinomial one can also be used in binary case, we still
implement a specialized
         * binary version for performance reason.
         */
        val margin = -1.0 * dot(data, weights)
        val multiplier = (1.0 / (1.0 + math.exp(margin))) - label
        axpy(multiplier, data, cumGradient)
        if (label > 0) {
          // The following is equivalent to log(1 + exp(margin)) but more
numerically stable.
          MLUtils.log1pExp(margin)
        } else {
          MLUtils.log1pExp(margin) - margin
        }
      case _ =>
        /**
         * For Multinomial Logistic Regression.
         */
        val weightsArray = weights match {
          case dv: DenseVector => dv.values
          case _ =>
            thrownew IllegalArgumentException(
              s"weights only supports dense vector but got type
${weights.getClass}.")
        }
        val cumGradientArray = cumGradient match {
          case dv: DenseVector => dv.values
          case _ =>
            thrownew IllegalArgumentException(
              s"cumGradient only supports dense vector but got type
${cumGradient.getClass}.")
```

```
        }

        // marginY is margins(label - 1) in the formula.
        var marginY = 0.0
        var maxMargin = Double.NegativeInfinity
        var maxMarginIndex = 0

        val margins = Array.tabulate(numClasses - 1) { i =>
          var margin = 0.0
          data.foreachActive { (index, value) =>
            if (value != 0.0) margin += value * weightsArray((i * dataSize) +
index)
          }
          if (i == label.toInt - 1) marginY = margin
          if (margin > maxMargin) {
            maxMargin = margin
            maxMarginIndex = i
          }
          margin
        }

        /**
         * When maxMargin > 0, the original formula will cause overflow as we
discuss
         * in the previous comment.
         * We address this by subtracting maxMargin from all the margins, so
it's guaranteed
         * that all of the new margins will be smaller than zero to prevent
arithmetic overflow.
         */
        val sum = {
          var temp = 0.0
          if (maxMargin > 0) {
            for (i <- 0 until numClasses - 1) {
              margins(i) -= maxMargin
              if (i == maxMarginIndex) {
                temp += math.exp(-maxMargin)
              } else {
                temp += math.exp(margins(i))
```

```
              }
            }
          } else {
            for (i <- 0 until numClasses - 1) {
              temp += math.exp(margins(i))
            }
          }
          temp
        }

        for (i <- 0 until numClasses - 1) {
          val multiplier = math.exp(margins(i)) / (sum + 1.0) - {
            if (label != 0.0 && label == i + 1) 1.0else0.0
          }
          data.foreachActive { (index, value) =>
            if (value != 0.0) cumGradientArray(i * dataSize + index) +=
multiplier * value
          }
        }

        val loss = if (label > 0.0) math.log1p(sum) -
marginY else math.log1p(sum)

        if (maxMargin > 0) {
          loss + maxMargin
        } else {
          loss
        }
    }
  }
}
```

（2）updater

```
//weihtsOld:上一次迭代计算后的特征权重向量
//gradient:本次迭代计算的特征权重向量
//stepSize:迭代步长
//iter:当前迭代次数
//regParam:正则参数
//以当前迭代次数的平方根的倒数作为本次迭代趋近(下降)的因子
```

```
//返回本次剃度下降后更新的特征权重向量
//使用了 L2 regularization (R(w) = 1/2 ||w||^2), weights 更新规则为:
```

$$weight := weight - \frac{stepSize}{\sqrt{iter}} * (gradient + regParam * weight)$$

```
/**
 * :: DeveloperApi ::
 * Updater for L2 regularized problems.
 *          R(w) = 1/2 ||w||^2
 * Uses a step-size decreasing with the square root of the number of
iterations.
 */
@DeveloperApi
class SquaredL2Updater extends Updater {
  overridedef compute(
      weightsOld: Vector,
      gradient: Vector,
      stepSize: Double,
      iter: Int,
      regParam: Double): (Vector, Double) = {
    // add up both updates from the gradient of the loss (= step) as well as
    // the gradient of the regularizer (= regParam * weightsOld)
    // w' = w - thisIterStepSize * (gradient + regParam * w)
    // w' = (1 - thisIterStepSize * regParam) * w - thisIterStepSize *
gradient
    val thisIterStepSize = stepSize / math.sqrt(iter)
    val brzWeights: BV[Double] = weightsOld.toBreeze.toDenseVector
    brzWeights :*= (1.0 - thisIterStepSize * regParam)
    brzAxpy(-thisIterStepSize, gradient.toBreeze, brzWeights)
    val norm = brzNorm(brzWeights, 2.0)

    (Vectors.fromBreeze(brzWeights), 0.5 * regParam * norm * norm)
  }
}
```

5.5 Spark MLlib 朴素贝叶斯分类算法

贝叶斯公式，或者叫做贝叶斯定理，是贝叶斯分类的基础。而贝叶斯分类是一类分类算法的统称，这一类算法的基础都是贝叶斯公式。目前研究较多的 4 种贝叶斯分类算法有：

Naive Bayes、TAN、BAN 和 GBN。

5.5.1 朴素贝叶斯分类算法

理工科的学生在大学应该都学过概率论，其中最重要的几个公式中就有贝叶斯公式——用来描述两个条件概率之间的关系，比如 P(A|B)和 P(B|A)。如何在已知事件 A 和 B 分别发生的概率和事件 B 发生时事件 A 发生的概率时求得事件 A 发生时事件 B 发生的概率，这就是贝叶斯公式的作用。其表述如下：

$$P(B|A)=P(A|B)\times P(B)P(A) \tag{5-18}$$

朴素贝叶斯分类（Naive Bayes）也可以叫 NB 算法。其核心思想非常简单：对于某一预测项，分别计算该预测项为各个分类的概率，然后选择概率最大的分类为其预测分类。就好像你预测一个娘炮是女人的可能性是 40%，是男人的可能性是 41%，那么就可以判断他是男人。

Naive Bayes 的数学定义如下：

（1）设 $x=\{a_1,a_2,..,a_m\}$ 为一个待分类项，而每个 a_i 为 x 的一个特征属性。
（2）已知类别集合 $C=\{y_1,y_2,..,y_n\}$。
（3）计算 xx 为各个类别的概率： $P(y_1|x),P(y_2|x),..,P(y_n|x)$。
（4）如果 $P(y_k|x)=\max\{P(y_1|x),P(y_2|x),..,P(y_n|x)\}$，则 x 的类别为 y_k。

如何获取第 4 步中的最大值，也就是如何计算第 3 步中的各个条件概率最为重要。可以采用如下做法：

（1）获取训练数据集，即分类已知的数据集。
（2）统计得到在各类别下各个特征属性的条件概率估计，即：

$P(a1|y1),P(a2|y1),...,P(a_m|y1);P(a1|y2),P(a2|y2),...,P(a_m|y2);...;P(a1|y_n),P(a2|y_n),...,P(a_m|y_n)$，其中的数据可以是离散的，也可以是连续的。

如果各个特征属性是条件独立的，则根据贝叶斯定理有如下推导：

$$P(y_i|x)=\frac{P(x|y_i)P(y_i)}{P(x)} \tag{5-19}$$

对于某 x 来说，分母是固定的，所以只要找出分子最大的即为条件概率最大的。又因为各特征属性是条件独立的，所以有：

$$P(x|y_i)P(y_i)=P(a_1|y_i)P(a_2|y_i)\ldots P(a_m|y_i)P(y_i)=P(y_i)\prod_{j=1}^{m}P(a_j|y_i) \tag{5-20}$$

在 Spark 源码中的计算概率 P 是在 NaiveBayes 类下面的 run 方法下，源码如下(Java)：

```
def run(data: RDD[LabeledPoint]): NaiveBayesModel = {
  val requireNonnegativeValues: Vector => Unit = (v: Vector) => {
```

```
      val values = v match {
        case sv: SparseVector => sv.values
        case dv: DenseVector => dv.values
      }
      if (!values.forall(_ >= 0.0)) {
        throw new SparkException(s"Naive Bayes requires nonnegative feature
values but found $v.")
      }
    }

    val requireZeroOneBernoulliValues: Vector => Unit = (v: Vector) => {
      val values = v match {
        case sv: SparseVector => sv.values
        case dv: DenseVector => dv.values
      }
      if (!values.forall(v => v == 0.0 || v == 1.0)) {
        throw new SparkException(
          s"Bernoulli naive Bayes requires 0 or 1 feature values but found $v.")
      }
    }

//对每个标签进行聚合操作（Aggregates term frequencies per label）.
    // TODO: Calling combineByKey and collect creates two stages, we can
implement something
    // TODO: similar to reduceByKeyLocally to save one stage.
    val aggregated = data.map(p => (p.label, p.features)).combineByKey[(Long,
DenseVector)](
     //检测方法
      createCombiner = (v: Vector) => {
        if (modelType == Bernoulli) {
          requireZeroOneBernoulliValues(v)
        } else {
          requireNonnegativeValues(v)
        }
        (1L, v.copy.toDense)
      },
      mergeValue = (c: (Long, DenseVector), v: Vector) => {
        requireNonnegativeValues(v)
        BLAS.axpy(1.0, v, c._2) //c.2 +=  1.0*v
```

```
      (c._1 + 1L, c._2)
    },
    mergeCombiners = (c1: (Long, DenseVector), c2: (Long, DenseVector)) => {
      BLAS.axpy(1.0, c2._2, c1._2) ////c.2 +=  1.0*c2._2
      (c1._1 + c2._1, c1._2)
    }
  ).collect().sortBy(_._1)

//标签数量
  val numLabels = aggregated.length
//聚合文档数量
  var numDocuments = 0L
  aggregated.foreach { case (_, (n, _)) =>
    numDocuments += n
  }
//特征数量
  val numFeatures = aggregated.head match { case (_, (_, v)) => v.size }

  val labels = new Array[Double](numLabels)
//建立一个 Array 来存储 pi 类别的先验概率
  val pi = new Array[Double](numLabels)
//特征下的条件概率
  val theta = Array.fill(numLabels)(new Array[Double](numFeatures))


  val piLogDenom = math.log(numDocuments + numLabels * lambda)
  var i = 0
  aggregated.foreach { case (label, (n, sumTermFreqs)) =>
    labels(i) = label
    pi(i) = math.log(n + lambda) - piLogDenom
    val thetaLogDenom = modelType match {
      case Multinomial => math.log(sumTermFreqs.values.sum + numFeatures *
lambda)
      case Bernoulli => math.log(n + 2.0 * lambda)
      case _ =>
        // This should never happen.
        throw new UnknownError(s"Invalid modelType: $modelType.")
    }
    var j = 0
```

```
      while (j < numFeatures) {
        theta(i)(j) = math.log(sumTermFreqs(j) + lambda) - thetaLogDenom
        j += 1
      }
      i += 1
    }
//返回一个 NaiveBayesModel，这个模型输入参数包含标签 labels，先验概率 pi，条件概率
theta ，模型方法 modelType（仅支持 Multinomial、Bernoulli ）
    new NaiveBayesModel(labels, pi, theta, modelType)
  }
}
```

对于类别集合 $C=\{y1,y2,\ldots,y_o\}$，它的先验概率是：

$$P_i(i)=\log\frac{n+lamda}{numDocuments+numLabels*lambda} \tag{5-21}$$

- lamda：平滑因子。
- numDoucuments：总的次数。
- numlaebls：类别数。

其中 thetaLogDenom 有两种模式：

（1）多项式模式：

```
case Multinomial => math.log(sumTermFreqs.values.sum + numFeatures * lambda)
```

（2）伯努利模式：

```
case Bernoulli => math.log(n + 2.0 * lambda)
```

n 为解释类别 y_i 的总数，那么 theta(i)(j):

$$theta(i)(j)=\log(P(a_j/y_j))=\begin{cases}\log\dfrac{sumTermFreqs(j)+lambda}{sumTermFreqs.values.sum+numFeatures*lambda}\\ \log\dfrac{sumTermFreqs(j)+lambda}{n+2.0*lambda}\end{cases} \tag{5-22}$$

5.5.2　朴素贝叶斯 Spark MLlib 源码

```
/**
 * 朴素贝叶斯分类器模型
 *
```

```
 * @param labels 标签
 * @param pi 对先验概率取 log 之后的值，维度与类别集合 C 的维度一样
 * @param theta 条件概率取 log，它的维度是 C-by-D D 是特征数
 * @param modelType 模型的模式类型，选择可以是 "multinomial"或"bernoulli"
 */
@Since("0.9.0")
class NaiveBayesModel private[spark] (
    @Since("1.0.0") val labels: Array[Double],
    @Since("0.9.0") val pi: Array[Double],
    @Since("0.9.0") val theta: Array[Array[Double]],
    @Since("1.4.0") val modelType: String)
  extends ClassificationModel with Serializable with Saveable {

  import NaiveBayes.{Bernoulli, Multinomial, supportedModelTypes}

  private val piVector = new DenseVector(pi)
  private val thetaMatrix = new DenseMatrix(labels.length, theta(0).length,
theta.flatten, true)

  private[mllib] def this(labels: Array[Double], pi: Array[Double], theta:
Array[Array[Double]]) =
    this(labels, pi, theta, NaiveBayes.Multinomial)

  /** A Java-friendly constructor that takes three Iterable parameters. */
  private[mllib] def this(
      labels: JIterable[Double],
      pi: JIterable[Double],
      theta: JIterable[JIterable[Double]]) =
    this(labels.asScala.toArray, pi.asScala.toArray,
theta.asScala.toArray.map(_.asScala.toArray))

  require(supportedModelTypes.contains(modelType),
    s"Invalid modelType $modelType. Supported modelTypes are
$supportedModelTypes.")

//Bernoulli 概率得分（大小）要求，当为1时，那么为 log(condprob)，当为0时，那么为 log(1-
condprob)//这个预计算的 log(1.0 - exp(theta))和它们的和是应用于这种情况下的预测
  //This precomputes log(1.0 - exp(theta)) and its sum which are used for the
linear algebra
```

```
// application of this condition (in predict function).
private val (thetaMinusNegTheta, negThetaSum) = modelType match {
  case Multinomial => (None, None)
  case Bernoulli =>
    val negTheta = thetaMatrix.map(value => math.log(1.0 - math.exp(value)))
    val ones = new DenseVector(Array.fill(thetaMatrix.numCols) {1.0})
    val thetaMinusNegTheta = thetaMatrix.map { value =>
      value - math.log(1.0 - math.exp(value))
    }
    (Option(thetaMinusNegTheta), Option(negTheta.multiply(ones)))
  case _ =>
    // This should never happen.
    throw new UnknownError(s"Invalid modelType: $modelType.")
}

@Since("1.0.0")
override def predict(testData: RDD[Vector]): RDD[Double] = {
  val bcModel = testData.context.broadcast(this)
  testData.mapPartitions { iter =>
    val model = bcModel.value
    iter.map(model.predict)
  }
}

@Since("1.0.0")
override def predict(testData: Vector): Double = {
  modelType match {
    case Multinomial =>
      labels(multinomialCalculation(testData).argmax)
    case Bernoulli =>
      labels(bernoulliCalculation(testData).argmax)
  }
}

/**
 * 输入数据，根据模型，进行预测
 *
 * @param testData 用 RDD 表示的数据，用于预测
 * @return an RDD[Vector] 预测返回值
```

```
 */
@Since("1.5.0")
def predictProbabilities(testData: RDD[Vector]): RDD[Vector] = {
  val bcModel = testData.context.broadcast(this)
  testData.mapPartitions { iter =>
    val model = bcModel.value
    iter.map(model.predictProbabilities)
  }
}

/**
* 使用该模型训练的一个单一数据点的后验概率。
 * Predict posterior class probabilities for a single data point using the
model trained.
 *
 * @param testData 用 RDD 表示的数据，用于预测
 * @return an RDD[Vector] 预测返回值
 */
@Since("1.5.0")
def predictProbabilities(testData: Vector): Vector = {
  modelType match {
    case Multinomial =>
      posteriorProbabilities(multinomialCalculation(testData))
    case Bernoulli =>
      posteriorProbabilities(bernoulliCalculation(testData))
  }
}

private def multinomialCalculation(testData: Vector) = {
  val prob = thetaMatrix.multiply(testData)
  BLAS.axpy(1.0, piVector, prob)
  prob
}

private def bernoulliCalculation(testData: Vector) = {
  testData.foreachActive((_, value) =>
    if (value != 0.0 && value != 1.0) {
      throw new SparkException(
        s"Bernoulli naive Bayes requires 0 or 1 feature values but found
```

```
$testData.")
      }
    )
    val prob = thetaMinusNegTheta.get.multiply(testData)
    BLAS.axpy(1.0, piVector, prob)
    BLAS.axpy(1.0, negThetaSum.get, prob)
    prob
  }

  private def posteriorProbabilities(logProb: DenseVector) = {
    val logProbArray = logProb.toArray
    val maxLog = logProbArray.max
    val scaledProbs = logProbArray.map(lp => math.exp(lp - maxLog))
    val probSum = scaledProbs.sum
    new DenseVector(scaledProbs.map(_ / probSum))
  }

  @Since("1.3.0")
  override def save(sc: SparkContext, path: String): Unit = {
    val data = NaiveBayesModel.SaveLoadV2_0.Data(labels, pi, theta, modelType)
    NaiveBayesModel.SaveLoadV2_0.save(sc, path, data)
  }

  override protected def formatVersion: String = "2.0"
}

@Since("1.3.0")
object NaiveBayesModel extends Loader[NaiveBayesModel] {

  import org.apache.spark.mllib.util.Loader._

  private[mllib] object SaveLoadV2_0 {

    def thisFormatVersion: String = "2.0"

    /** Hard-code class name string in case it changes in the future */
    def thisClassName: String =
"org.apache.spark.mllib.classification.NaiveBayesModel"
```

```
    /** Model data for model import/export */
    case class Data(
        labels: Array[Double],
        pi: Array[Double],
        theta: Array[Array[Double]],
        modelType: String)

    def save(sc: SparkContext, path: String, data: Data): Unit = {
      val sqlContext = SQLContext.getOrCreate(sc)
      import sqlContext.implicits._

      // Create JSON metadata.
      val metadata = compact(render(
        ("class" -> thisClassName) ~ ("version" -> thisFormatVersion) ~
          ("numFeatures" -> data.theta(0).length) ~ ("numClasses" ->
data.pi.length)))
      sc.parallelize(Seq(metadata), 1).saveAsTextFile(metadataPath(path))

      // Create Parquet data.
      val dataRDD: DataFrame = sc.parallelize(Seq(data), 1).toDF()
      dataRDD.write.parquet(dataPath(path))
    }

    @Since("1.3.0")
    def load(sc: SparkContext, path: String): NaiveBayesModel = {
      val sqlContext = SQLContext.getOrCreate(sc)
      // Load Parquet data.
      val dataRDD = sqlContext.read.parquet(dataPath(path))
      // Check schema explicitly since erasure makes it hard to use match-case
for checking.
      checkSchema[Data](dataRDD.schema)
      val dataArray = dataRDD.select("labels", "pi", "theta",
"modelType").take(1)
      assert(dataArray.length == 1, s"Unable to load NaiveBayesModel data from:
${dataPath(path)}")
      val data = dataArray(0)
      val labels = data.getAs[Seq[Double]](0).toArray
      val pi = data.getAs[Seq[Double]](1).toArray
      val theta = data.getAs[Seq[Seq[Double]]](2).map(_.toArray).toArray
```

```
    val modelType = data.getString(3)
    new NaiveBayesModel(labels, pi, theta, modelType)
  }

 }

 private[mllib] object SaveLoadV1_0 {

   def thisFormatVersion: String = "1.0"

/** Hard-code class name string in case it changes in the future */
   def thisClassName: String =
"org.apache.spark.mllib.classification.NaiveBayesModel"

/** 模型数据的导入导出 Model data for model import/export */
   case class Data(
       labels: Array[Double],
       pi: Array[Double],
       theta: Array[Array[Double]])

   def save(sc: SparkContext, path: String, data: Data): Unit = {
     val sqlContext = SQLContext.getOrCreate(sc)
     import sqlContext.implicits._

//建立一个 JSON 文件数据 Create JSON metadata.
     val metadata = compact(render(
       ("class" -> thisClassName) ~ ("version" -> thisFormatVersion) ~
         ("numFeatures" -> data.theta(0).length) ~ ("numClasses" ->
data.pi.length)))
     sc.parallelize(Seq(metadata), 1).saveAsTextFile(metadataPath(path))

// 建立一个 Parquet 文件数据 Create Parquet data.
     val dataRDD: DataFrame = sc.parallelize(Seq(data), 1).toDF()
     dataRDD.write.parquet(dataPath(path))
   }

   def load(sc: SparkContext, path: String): NaiveBayesModel = {
     val sqlContext = SQLContext.getOrCreate(sc)
     // Load Parquet data.
```

```
      val dataRDD = sqlContext.read.parquet(dataPath(path))
      // Check schema explicitly since erasure makes it hard to use match-case
for checking.
      checkSchema[Data](dataRDD.schema)
      val dataArray = dataRDD.select("labels", "pi", "theta").take(1)
      assert(dataArray.length == 1, s"Unable to load NaiveBayesModel data from:
${dataPath(path)}")
      val data = dataArray(0)
      val labels = data.getAs[Seq[Double]](0).toArray
      val pi = data.getAs[Seq[Double]](1).toArray
      val theta = data.getAs[Seq[Seq[Double]]](2).map(_.toArray).toArray
      new NaiveBayesModel(labels, pi, theta)
    }
  }

  override def load(sc: SparkContext, path: String): NaiveBayesModel = {
    val (loadedClassName, version, metadata) = loadMetadata(sc, path)
    val classNameV1_0 = SaveLoadV1_0.thisClassName
    val classNameV2_0 = SaveLoadV2_0.thisClassName
    val (model, numFeatures, numClasses) = (loadedClassName, version) match {
      case (className, "1.0") if className == classNameV1_0 =>
        val (numFeatures, numClasses) =
ClassificationModel.getNumFeaturesClasses(metadata)
        val model = SaveLoadV1_0.load(sc, path)
        (model, numFeatures, numClasses)
      case (className, "2.0") if className == classNameV2_0 =>
        val (numFeatures, numClasses) =
ClassificationModel.getNumFeaturesClasses(metadata)
        val model = SaveLoadV2_0.load(sc, path)
        (model, numFeatures, numClasses)
      case _ => throw new Exception(
        s"NaiveBayesModel.load did not recognize model with (className, format
version):" +
        s"($loadedClassName, $version).  Supported:\n" +
        s"  ($classNameV1_0, 1.0)")
    }
    assert(model.pi.length == numClasses,
      s"NaiveBayesModel.load expected $numClasses classes," +
        s" but class priors vector pi had ${model.pi.length} elements")
```

```
    assert(model.theta.length == numClasses,
      s"NaiveBayesModel.load expected $numClasses classes," +
        s" but class conditionals array theta had ${model.theta.length}
elements")
    assert(model.theta.forall(_.length == numFeatures),
      s"NaiveBayesModel.load expected $numFeatures features," +
        s" but class conditionals array theta had elements of size:" +
        s" ${model.theta.map(_.length).mkString(",")}")
    model
  }
}

/**
 * 输入一个 RDD （(label, features)` pairs）来训练朴素贝叶斯模型
 *
 * This is the Multinomial NB ([[http://tinyurl.com/lsdw6p]]) which can handle
all kinds of
 * discrete data.  For example, by converting documents into TF-IDF vectors, it
can be used for
 * document classification.  By making every vector a 0-1 vector, it can also
be used as
 * Bernoulli NB ([[http://tinyurl.com/p7c96j6]]). The input feature values
must be nonnegative.
 */
@Since("0.9.0")
class NaiveBayes private (
    private var lambda: Double,
    private var modelType: String) extends Serializable with Logging {

  import NaiveBayes.{Bernoulli, Multinomial}

  @Since("1.4.0")
  def this(lambda: Double) = this(lambda, NaiveBayes.Multinomial)

  @Since("0.9.0")
  def this() = this(1.0, NaiveBayes.Multinomial)

/** 设置平滑参数（因子）Set the smoothing parameter. Default: 1.0. */
  @Since("0.9.0")
```

```
  def setLambda(lambda: Double): NaiveBayes = {
    require(lambda >= 0,
      s"Smoothing parameter must be nonnegative but got ${lambda}")
    this.lambda = lambda
    this
  }

/** 得到平滑参数（因子）Get the smoothing parameter. */
  @Since("1.4.0")
  def getLambda: Double = lambda

 /**
   * 设置模型的模式类型，"multinomial" (default) and "bernoulli".
   * Set the model type using a string (case-sensitive).
   * Supported options: "multinomial" (default) and "bernoulli".
   */
  @Since("1.4.0")
  def setModelType(modelType: String): NaiveBayes = {
    require(NaiveBayes.supportedModelTypes.contains(modelType),
      s"NaiveBayes was created with an unknown modelType: $modelType.")
    this.modelType = modelType
    this
  }

  /** 得到模式类型 Get the model type. */
  @Since("1.4.0")
  def getModelType: String = this.modelType

 /**
   * Run the algorithm with the configured parameters on an input RDD of
LabeledPoint entries.
   *
   * @param data RDD of [[org.apache.spark.mllib.regression.LabeledPoint]].
   */
def run(data: RDD[LabeledPoint]): NaiveBayesModel = {
    val requireNonnegativeValues: Vector => Unit = (v: Vector) => {
      val values = v match {
        case sv: SparseVector => sv.values
        case dv: DenseVector => dv.values
```

```
      }
      if (!values.forall(_ >= 0.0)) {
        throw new SparkException(s"Naive Bayes requires nonnegative feature
values but found $v.")
      }
    }

    val requireZeroOneBernoulliValues: Vector => Unit = (v: Vector) => {
      val values = v match {
        case sv: SparseVector => sv.values
        case dv: DenseVector => dv.values
      }
      if (!values.forall(v => v == 0.0 || v == 1.0)) {
        throw new SparkException(
          s"Bernoulli naive Bayes requires 0 or 1 feature values but found $v.")
      }
    }

// 对每个标签进行聚合操作（Aggregates term frequencies per label）.
// TODO: Calling combineByKey and collect creates two stages, we can implement
something
// TODO: similar to reduceByKeyLocally to save one stage.
    val aggregated = data.map(p => (p.label, p.features)).combineByKey[(Long,
DenseVector)](
     //检测方法
      createCombiner = (v: Vector) => {
        if (modelType == Bernoulli) {
          requireZeroOneBernoulliValues(v)
        } else {
          requireNonnegativeValues(v)
        }
        (1L, v.copy.toDense)
      },
      mergeValue = (c: (Long, DenseVector), v: Vector) => {
        requireNonnegativeValues(v)
        BLAS.axpy(1.0, v, c._2) //c.2 +=  1.0*v
        (c._1 + 1L, c._2)
      },
      mergeCombiners = (c1: (Long, DenseVector), c2: (Long, DenseVector)) => {
```

```
      BLAS.axpy(1.0, c2._2, c1._2) ////c.2 +=  1.0*c2._2
      (c1._1 + c2._1, c1._2)
    }
  ).collect().sortBy(_._1)

   //标签数量
  val numLabels = aggregated.length
   //聚合文档数量
  var numDocuments = 0L
  aggregated.foreach { case (_, (n, _)) =>
    numDocuments += n
  }
   //特征数量
  val numFeatures = aggregated.head match { case (_, (_, v)) => v.size }

  val labels = new Array[Double](numLabels)
   //建立一个 Array 来存储 pi 类别的先验概率
  val pi = new Array[Double](numLabels)
   //特征下的条件概率
  val theta = Array.fill(numLabels)(new Array[Double](numFeatures))


  val piLogDenom = math.log(numDocuments + numLabels * lambda)
  var i = 0
  aggregated.foreach { case (label, (n, sumTermFreqs)) =>
    labels(i) = label
    pi(i) = math.log(n + lambda) - piLogDenom
    val thetaLogDenom = modelType match {
      case Multinomial => math.log(sumTermFreqs.values.sum + numFeatures *
lambda)
      case Bernoulli => math.log(n + 2.0 * lambda)
      case _ =>
        // This should never happen.
        throw new UnknownError(s"Invalid modelType: $modelType.")
    }
    var j = 0
    while (j < numFeatures) {
      theta(i)(j) = math.log(sumTermFreqs(j) + lambda) - thetaLogDenom
      j += 1
```

```
      }
      i += 1
    }
//返回一个 NaiveBayesModel，这个模型包含标签 labels，先验概率 pi，条件概率 theta ，模型方
法 modelType（仅支持 Multinomial、Bernoulli ）
    new NaiveBayesModel(labels, pi, theta, modelType)
  }
}

/**
 * Top-level methods for calling naive Bayes.
 */
@Since("0.9.0")
object NaiveBayes {

  /** String name for multinomial model type. */
  private[spark] val Multinomial: String = "multinomial"

  /** String name for Bernoulli model type. */
  private[spark] val Bernoulli: String = "bernoulli"

  /* Set of modelTypes that NaiveBayes supports */
  private[spark] val supportedModelTypes = Set(Multinomial, Bernoulli)

  /**
   * Trains a Naive Bayes model given an RDD of `(label, features)` pairs.
   *
   * This is the default Multinomial NB ([[http://tinyurl.com/lsdw6p]]) which
can handle all
   * kinds of discrete data.  For example, by converting documents into TF-IDF
vectors, it
   * can be used for document classification.
   *
   * This version of the method uses a default smoothing parameter of 1.0.
   *
   * @param input RDD of `(label, array of features)` pairs.  Every vector
should be a frequency
   *              vector or a count vector.
   */
```

```
  @Since("0.9.0")
  def train(input: RDD[LabeledPoint]): NaiveBayesModel = {
    new NaiveBayes().run(input)
  }

  /**
   * Trains a Naive Bayes model given an RDD of `(label, features)` pairs.
   *
   * This is the default Multinomial NB ([[http://tinyurl.com/lsdw6p]]) which
can handle all
   * kinds of discrete data.  For example, by converting documents into TF-IDF
vectors, it
   * can be used for document classification.
   *
   * @param input RDD of `(label, array of features)` pairs.  Every vector
should be a frequency
   *              vector or a count vector.
   * @param lambda The smoothing parameter
   */
  @Since("0.9.0")
  def train(input: RDD[LabeledPoint], lambda: Double): NaiveBayesModel = {
    new NaiveBayes(lambda, Multinomial).run(input)
  }

  /**
   * Trains a Naive Bayes model given an RDD of `(label, features)` pairs.
   *
   * The model type can be set to either Multinomial NB
([[http://tinyurl.com/lsdw6p]])
   * or Bernoulli NB ([[http://tinyurl.com/p7c96j6]]). The Multinomial NB can
handle
   * discrete count data and can be called by setting the model type to
"multinomial".
   * For example, it can be used with word counts or TF_IDF vectors of
documents.
   * The Bernoulli model fits presence or absence (0-1) counts. By making every
vector a
   * 0-1 vector and setting the model type to "bernoulli", the  fits and
predicts as
```

```
   * Bernoulli NB.
   *
   * @param input RDD of `(label, array of features)` pairs.  Every vector
should be a frequency
   *              vector or a count vector.
   * @param lambda The smoothing parameter
   *
   * @param modelType The type of NB model to fit from the enumeration
NaiveBayesModels, can be
   * multinomial or bernoulli
   */
  @Since("1.4.0")
  def train(input: RDD[LabeledPoint], lambda: Double, modelType: String):
NaiveBayesModel = {
    require(supportedModelTypes.contains(modelType),
      s"NaiveBayes was created with an unknown modelType: $modelType.")
    new NaiveBayes(lambda, modelType).run(input)
  }

}
```

5.6 Spark MLlib 决策树算法

决策树是常用的分类算法之一，其对于探索式的知识发现往往有较好的表现。决策树原理十分简单，可处理大维度的数据，不用预先对模型的特征有所了解，这些特性使得决策树被广泛使用。决策树采用贪心算法，其建立过程同样需要训练数据。决策树的核心问题是决策树分支准则的确定，以及分裂点的确定。

5.6.1 决策树算法

决策树作为一种分类回归算法，在处理非线性、特征值缺少的数据方面有很多的优势，能够处理不相干的特征，并且对分类的结果通过树的方式有比较清晰的结构解释，但是容易过拟合，针对这个问题，可以采取对树进行剪枝的方式，还有一些融合集成的解决方案，比如随机森林（RandomForest）、GBDT （Gradient Boost Decision Tree）等。

模型的训练过程其实是决策树的构造过程，它采用自顶向下的递归方式，在决策树的内部结点进行属性值的比较，并根据不同的属性值判断从该结点向下分支，进行递归划分，直到满足一定的终止条件（可以进行自定义），其中叶结点是要学习划分的类。在当前节点用哪个属性特征作为判断进行切分（也叫分裂规则），取决于切分后节点数据集合中的类别

（分区）的有序（纯）程度，划分后的分区数据越纯，那么当前分裂规则也越合适。衡量节点数据集合的有序无序性，有熵、基尼（Gini）、方差 3 种，其中熵和 Gini 是针对分类的，方差是针对回归的[83]。两种杂质的分类方法（Gini 和熵）和测量回归（方差）如表 5-11 所示。

表 5-11　两种杂质的分类方法（Gini 和熵）和测量回归（方差）

Impurity	Task	Formula	Description
Gini impurity	Classsification	$\sum_{i=1}^{C} f_i(1-f_i)$	f_i is the frequency of label i at a node and C is the number of unique labels
Entropy	Classsification	$\sum_{i=1}^{C} -f_i \log(f_i)$	f_i is the frequency of label i at a node and C is the number of unique labels
Variance	Regression	$\frac{1}{N}\sum_{i=1}^{N}(y_i-\mu)^2$	y_i is label for an instance, N is the number of instances and μ is the mean given by $\frac{1}{N}\sum_{i=1}^{N} y_i$

信息熵是信息论中的基本概念。信息论是 C.E.Shannon 于 1948 年提出并由此发展起来的，主要用于解决信息传递过程中的问题，也称为统计通信理论。信息论认为：信息是用来消除随机不确定性的，信息量的大小可由所消除的不确定大小来计量。

熵代表集合的无序性的参数，熵越大，代表越无序、越不纯。熵的公式如下：

$$Entropy(S)=\sum_{i=1}^{C}-P_i\log_2 P_i \tag{5-23}$$

其中 C 表示类别，是样本集合中属于类别 i 的概率。

在决策树分类中，一般是用信息增益 infoGain 来作为决策树节点特征属性划分的依据，采用使得信息增益最大的属性作为数据划分的度量依赖。信息增益 infoGain 定义如下：

$$Gain(S,A)=Entropy(S)-\sum_{v\in V(A)}\frac{|S_v|}{|S|}Entropy(S_v) \tag{5-24}$$

其中 $V(A)$代表属性 A 的分区，S 代表样本集合，S_v 是 S 中属性 A 的值属于 v 分区的样本集合。

决策树的算法实现在学术界有 ID3、C4.5、CART 等， ID3 采用信息增益作为属性选择的度量，参见上面的 Gain，这种方式的一个缺点是在计算信息增益时，倾向于选择具有大量值的属性，因此提出了 C4.5 的基于信息增益率的度量，而 CART 使用基尼（Gini）指数作为属性选择的度量，这些算法之间的差别主要包括在训练创建决策树过程中如何选择属性特征，以及剪枝的机制处理。

Spark MLlib 对决策树提供了二元以及多 label 的分类以及回归的支持，支持连续型和离散型的特征变量。这里的决策树是一颗二叉树，因此信息增益 infoGain 就为：

$$Gain(S,A)=Entropy(S)-\frac{|S_{left}|}{|S|}Entropy(S_{left})-\frac{|S_{right}|}{|S|}Entropy(S_{right}) \qquad (5\text{-}25)$$

MLlib 中的 bin 和 split，split 为切割点，对应二叉的决策树；bin 为桶或者箱子数，一个 split 把数据集划分成 2 个桶，所以 bin 是 split 的 2 倍。

在决策树的训练时，需要两个重要的参数：

- maxBins：每个特征分裂时，最大划分(桶)数量。
- maxDepth：树的最大高度。

在 MLlib 中，基本的样本训练决策树的构建流程为：寻找所有特征的可能的划分 split 以及桶信息 bin，针对每次划分 split，在 spark executors 上计算每个样本应该属于哪一个 bin，后聚合每一个 bin 的统计信息，在 Drivers 上通过这些统计信息计算每次 split 的信息增益，并选择一个信息增益最大的分割 split，按照该 split 对当前节点进行分割，直到满足终止条件。

为了防止过拟合，需要考虑剪枝，这里采用的是前向剪枝，当任一以下情况发生，MLlib 的决策树节点就终止划分，形成叶子节点。

树高度达到 maxDepth：

- minInfoGain，当前节点的所有属性分割带来的信息增益都比这个值要小。
- minInstancesPerNode，需要保证节点分割出的左右子节点的最少的样本数量达到这个值。

停止准则：当树停止建设（添加新的节点）时，确定调整这些参数， 保持测试数据来避免过拟合验证。

- MAXDEPTH：一棵树的最大深度。深树更具有表现力（可能允许更高的精度），但也更容易过度拟合。
- mininstancespernode：一个节点被分裂，它的每一个儿子节点都必须接受至少这个数量的训练实例。因为经常比个体树木训练更深，所以常用随机森林。
- mininfogain：一个节点将进一步分裂必须在信息增益上提高。

可调参数：小心保持的试验数据，以避免过拟合验证调整。

- maxBins：离散化连续属性算法时，使用一定数量箱数。
- maxMemoryInMB：用于收集足够的数据存储量。
- subsamplingRate：用于学习的决策树训练数据的分数。培养一个决策树，这个参数是有用的，因为训练实例的数量一般不是主要的制约因素。
- impurity：杂质用于选择候选分割，这一措施必须匹配算法参数。

缓存和检查点：

- useNodeIdCache：如果设置为 true，该算法将避免过流模型（树或树林）减少每个迭代的执行者通信代价。

- checkpointDir：检查点节点 ID 缓存 RDDS 目录。
- checkpointInterval：检查点节点 ID 缓存 RDDS 频率。设置太低，会导致写 HDFS 额外的开销太高；如果执行失败，RDD 需要重新计算问题。

5.6.2 决策树实例

1. 分类

下面的示例演示如何加载 libsvm 数据文件， 解析为 RDDlabeledpoint，然后利用决策树进行分类 ，Gini 作为杂质测量和最大树深度为 5 测试误差来衡量算法的准确性。

（1）Scala

决策树文档和文档在 API 的细节。

```
importorg.apache.spark.mllib.tree.DecisionTree
importorg.apache.spark.mllib.tree.model.DecisionTreeModel
importorg.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
valdata=MLUtils.loadLibSVMFile(sc,"data/mllib/sample_libsvm_data.txt")
// Split the data into training and test sets (30% held out for testing)
valsplits=data.randomSplit(Array(0.7,0.3))
val(trainingData,testData)=(splits(0),splits(1))

// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
valnumClasses=2
valcategoricalFeaturesInfo=Map[Int, Int]()
valimpurity="gini"
valmaxDepth=5
valmaxBins=32

valmodel=DecisionTree.trainClassifier(trainingData,numClasses,categoricalFeatu
resInfo,
impurity,maxDepth,maxBins)

// Evaluate model on test instances and compute test error
vallabelAndPreds=testData.map{point=>
valprediction=model.predict(point.features)
(point.label,prediction)
}
```

```
valtestErr=labelAndPreds.filter(r=>r._1!=r._2).count().toDouble/testData.count
()
println("Test Error = "+testErr)
println("Learned classification tree model:\n"+model.toDebugString)

// Save and load model
model.save(sc,"target/tmp/myDecisionTreeClassificationModel")
valsameModel=DecisionTreeModel.load(sc,"target/tmp/myDecisionTreeClassificatio
nModel")
```

（2）Java

决策树 DecisionTree Java docs 和 DecisionTreeModel Java docs 文档在 API 的细节。

```
importjava.util.HashMap;
importjava.util.Map;
importscala.Tuple2;
importorg.apache.spark.SparkConf;
importorg.apache.spark.api.java.JavaPairRDD;
importorg.apache.spark.api.java.JavaRDD;
importorg.apache.spark.api.java.JavaSparkContext;
importorg.apache.spark.api.java.function.Function;
importorg.apache.spark.api.java.function.PairFunction;
importorg.apache.spark.mllib.regression.LabeledPoint;
importorg.apache.spark.mllib.tree.DecisionTree;
importorg.apache.spark.mllib.tree.model.DecisionTreeModel;
importorg.apache.spark.mllib.util.MLUtils;

SparkConfsparkConf=newSparkConf().setAppName("JavaDecisionTreeClassificationEx
ample");
JavaSparkContextjsc=newJavaSparkContext(sparkConf);

// Load and parse the data file.
Stringdatapath="data/mllib/sample_libsvm_data.txt";
JavaRDD<LabeledPoint>data=MLUtils.loadLibSVMFile(jsc.sc(),datapath).toJavaRDD(
);
// Split the data into training and test sets (30% held out for testing)
JavaRDD<LabeledPoint>[]splits=data.randomSplit(newdouble[]{0.7,0.3});
JavaRDD<LabeledPoint>trainingData=splits[0];
JavaRDD<LabeledPoint>testData=splits[1];
```

```
// Set parameters.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
IntegernumClasses=2;
Map<Integer,Integer>categoricalFeaturesInfo=newHashMap<>();
Stringimpurity="gini";
IntegermaxDepth=5;
IntegermaxBins=32;

// Train a DecisionTree model for classification.
finalDecisionTreeModelmodel=DecisionTree.trainClassifier(trainingData,numClass
es,
categoricalFeaturesInfo,impurity,maxDepth,maxBins);

// Evaluate model on test instances and compute test error
JavaPairRDD<Double,Double>predictionAndLabel=
testData.mapToPair(newPairFunction<LabeledPoint,Double,Double>(){
@Override
  publicTuple2<Double,Double>call(LabeledPointp){
    returnnewTuple2<>(model.predict(p.features()),p.label());
  }
});
DoubletestErr=
1.0*predictionAndLabel.filter(newFunction<Tuple2<Double,Double>,Boolean>(){
@Override
publicBooleancall(Tuple2<Double,Double>pl){
return!pl._1().equals(pl._2());
}
}).count()/testData.count();

System.out.println("Test Error: "+testErr);
System.out.println("Learned classification tree
model:\n"+model.toDebugString());

// Save and load model
model.save(jsc.sc(),"target/tmp/myDecisionTreeClassificationModel");
DecisionTreeModelsameModel=DecisionTreeModel
.load(jsc.sc(),"target/tmp/myDecisionTreeClassificationModel");
```

（3）Python

DecisionTree Python docs 和 DecisionTreeModel Python docs 文档在 API 的细节。

```
frompyspark.mllib.treeimportDecisionTree,DecisionTreeModel
frompyspark.mllib.utilimportMLUtils

# Load and parse the data file into an RDD of LabeledPoint.
data=MLUtils.loadLibSVMFile(sc,'data/mllib/sample_libsvm_data.txt')
# Split the data into training and test sets (30% held out for testing)
(trainingData,testData)=data.randomSplit([0.7,0.3])

# Train a DecisionTree model.
#  Empty categoricalFeaturesInfo indicates all features are continuous.
model=DecisionTree.trainClassifier(trainingData,numClasses=2,categoricalFeatur
esInfo={},
impurity='gini',maxDepth=5,maxBins=32)

# Evaluate model on test instances and compute test error
predictions=model.predict(testData.map(lambdax:x.features))
labelsAndPredictions=testData.map(lambdalp:lp.label).zip(predictions)
testErr=labelsAndPredictions.filter(lambda(v,p):v!=p).count()/float(testData.c
ount())
print('Test Error = '+str(testErr))
print('Learned classification tree model:')
print(model.toDebugString())

# Save and load model
model.save(sc,"target/tmp/myDecisionTreeClassificationModel")
sameModel=DecisionTreeModel.load(sc,"target/tmp/myDecisionTreeClassificationMo
del")
```

2. 回归

下面的示例演示如何加载 LIBSVM data 文件，解析为 RDD of LabeledPoint，然后使用决策树的方差作为执行回归措施。平均平方误差（MSE）计算最后评价拟合优度检验。

（1）Scala

DecisionTree Scala docs 和 DecisionTreeModel Scala docs 文档在 API 的细节。

```
importorg.apache.spark.mllib.tree.DecisionTree
importorg.apache.spark.mllib.tree.model.DecisionTreeModel
importorg.apache.spark.mllib.util.MLUtils
```

```
// Load and parse the data file.
valdata=MLUtils.loadLibSVMFile(sc,"data/mllib/sample_libsvm_data.txt")
// Split the data into training and test sets (30% held out for testing)
valsplits=data.randomSplit(Array(0.7,0.3))
val(trainingData,testData)=(splits(0),splits(1))

// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
valcategoricalFeaturesInfo=Map[Int, Int]()
valimpurity="variance"
valmaxDepth=5
valmaxBins=32

valmodel=DecisionTree.trainRegressor(trainingData,categoricalFeaturesInfo,impu
rity,
maxDepth,maxBins)

// Evaluate model on test instances and compute test error
vallabelsAndPredictions=testData.map{point=>
valprediction=model.predict(point.features)
(point.label,prediction)
}
valtestMSE=labelsAndPredictions.map{case(v,p)=>math.pow(v-p,2)}.mean()
println("Test Mean Squared Error = "+testMSE)
println("Learned regression tree model:\n"+model.toDebugString)

// Save and load model
model.save(sc,"target/tmp/myDecisionTreeRegressionModel")
valsameModel=DecisionTreeModel.load(sc,"target/tmp/myDecisionTreeRegressionMod
el")
```

（2）Java

DecisionTree Java docs 和 DecisionTreeModel Java docs 文档在 API 的细节。

```
importjava.util.HashMap;
importjava.util.Map;
importscala.Tuple2;
importorg.apache.spark.SparkConf;
importorg.apache.spark.api.java.JavaPairRDD;
```

```
importorg.apache.spark.api.java.JavaRDD;
importorg.apache.spark.api.java.JavaSparkContext;
importorg.apache.spark.api.java.function.Function;
importorg.apache.spark.api.java.function.Function2;
importorg.apache.spark.api.java.function.PairFunction;
importorg.apache.spark.mllib.regression.LabeledPoint;
importorg.apache.spark.mllib.tree.DecisionTree;
importorg.apache.spark.mllib.tree.model.DecisionTreeModel;
importorg.apache.spark.mllib.util.MLUtils;

SparkConfsparkConf=newSparkConf().setAppName("JavaDecisionTreeRegressionExampl
e");
JavaSparkContextjsc=newJavaSparkContext(sparkConf);

// Load and parse the data file.
Stringdatapath="data/mllib/sample_libsvm_data.txt";
JavaRDD<LabeledPoint>data=MLUtils.loadLibSVMFile(jsc.sc(),datapath).toJavaRDD(
);
// Split the data into training and test sets (30% held out for testing)
JavaRDD<LabeledPoint>[]splits=data.randomSplit(newdouble[]{0.7,0.3});
JavaRDD<LabeledPoint>trainingData=splits[0];
JavaRDD<LabeledPoint>testData=splits[1];

// Set parameters.
// Empty categoricalFeaturesInfo indicates all features are continuous.
Map<Integer,Integer>categoricalFeaturesInfo=newHashMap<>();
Stringimpurity="variance";
IntegermaxDepth=5;
IntegermaxBins=32;

// Train a DecisionTree model.
finalDecisionTreeModelmodel=DecisionTree.trainRegressor(trainingData,
categoricalFeaturesInfo,impurity,maxDepth,maxBins);

// Evaluate model on test instances and compute test error
JavaPairRDD<Double,Double>predictionAndLabel=
testData.mapToPair(newPairFunction<LabeledPoint,Double,Double>(){
@Override
  publicTuple2<Double,Double>call(LabeledPointp){
```

```
    returnnewTuple2<>(model.predict(p.features()),p.label());
  }
});
DoubletestMSE=
predictionAndLabel.map(newFunction<Tuple2<Double,Double>,Double>(){
@Override
  publicDoublecall(Tuple2<Double,Double>pl){
    Doublediff=pl._1()-pl._2();
    returndiff*diff;
  }
}).reduce(newFunction2<Double,Double,Double>(){
@Override
  publicDoublecall(Doublea,Doubleb){
    returna+b;
  }
})/data.count();
System.out.println("Test Mean Squared Error: "+testMSE);
System.out.println("Learned regression tree model:\n"+model.toDebugString());

// Save and load model
model.save(jsc.sc(),"target/tmp/myDecisionTreeRegressionModel");
DecisionTreeModelsameModel=DecisionTreeModel
.load(jsc.sc(),"target/tmp/myDecisionTreeRegressionModel");
```

（3）Python

DecisionTree Python docs 和 DecisionTreeModel Python docs 文档在 API 的细节。

```
frompyspark.mllib.treeimportDecisionTree,DecisionTreeModel
frompyspark.mllib.utilimportMLUtils

# Load and parse the data file into an RDD of LabeledPoint.
data=MLUtils.loadLibSVMFile(sc,'data/mllib/sample_libsvm_data.txt')
# Split the data into training and test sets (30% held out for testing)
(trainingData,testData)=data.randomSplit([0.7,0.3])

# Train a DecisionTree model.
#  Empty categoricalFeaturesInfo indicates all features are continuous.
model=DecisionTree.trainRegressor(trainingData,categoricalFeaturesInfo={},
impurity='variance',maxDepth=5,maxBins=32)
```

```
# Evaluate model on test instances and compute test error
predictions=model.predict(testData.map(lambdax:x.features))
labelsAndPredictions=testData.map(lambdalp:lp.label).zip(predictions)
testMSE=labelsAndPredictions.map(lambda(v,p):(v-p)*(v-p)).sum()/\
float(testData.count())
print('Test Mean Squared Error = '+str(testMSE))
print('Learned regression tree model:')
print(model.toDebugString())

# Save and load model
model.save(sc,"target/tmp/myDecisionTreeRegressionModel")
sameModel=DecisionTreeModel.load(sc,"target/tmp/myDecisionTreeRegressionModel")
```

5.7 Spark MLlib KMeans 聚类算法

聚类算法常被用于数据量非常大的应用中。我们都知道，机器学习算法大体分为三类：监督学习（supervised learning）、无监督学习（unsupervised learning）和半监督学习（semi-supervised learning）。

监督学习是指我们利用带有类别属性标注的数据去训练、学习，用于预测未知数据的类别属性。例如，根据用户之前的购物行为去预测用户是否会购买某一商品。常用的算法有决策树、支持向量机 SVM、朴素贝叶斯分类器、K-近邻算法 KNN、线性回归和逻辑回归等。

无监督学习是指在无人工干预的情况下将数据按照相似程度划分，而聚类算法就是非常典型的无监督学习方法，通常要处理的数据没有标签信息，可以通过计算数据之间的相似性来自动划分类别[84-86]。

5.7.1 KMeans 聚类算法

K-Means 算法的思想是初始随机给定 K 个簇中心，按照距离最近原则把待分类的样本点分到各个簇，然后按平均法重新计算各个簇的质心，从而确定新的簇心，迭代计算，直到簇心的移动距离小于某个给定的误差值。使用算法描述语言，只要 4 个步骤：

（1）任意选择 K 个点作为初始聚类中心。

（2）计算每个样本点到聚类中心的距离，将每个样本点划分到离该点最近的聚类中去。

（3）计算每个聚类中所有点的坐标平均值，并将这个平均值作为新的聚类中心。

（4）反复执行（2）（3），直到聚类中心的移动小于某误差值或者聚类次数达到要求为止。

这里计算距离的方法通常是计算欧几里得距离，假设中心点 center 是(x_1, y_1)，需要计算的样本点 point 是(x_2, y_2)。

另外，给出损失函数（Cost Function），每一次选取好新的中心点，我们就要计算一下

当前选好的中心点损失为多少，这个损失代表着偏移量，越大说明当前聚类的效果越差，计算公式称为（Within-Cluster Sum of Squares, WCSS）：

$$L(C)=\sum_{k\in K}\sum_{i\in K}\|x_i-c_k\|^2 \tag{5-26}$$

其中，x_i 表示某一对象，c_k 表示该对象所属类别的中心点。整个式子的含义就是对各个类别下的对象，求对象与中心点的欧式距离的平方，把所有的平方求和就是 $L(C)$。

5.7.2 Spark MLlib KMeans 源码分析

```
class KMeansprivate (
    privatevar k: Int,
    privatevar maxIterations: Int,
    privatevar runs: Int,
    privatevar initializationMode: String,
    privatevar initializationSteps: Int,
    privatevar epsilon: Double,
    privatevar seed: Long)extends Serializablewith Logging {
//KMeans 类参数：
k:聚类个数，默认2；maxIterations：迭代次数，默认20；runs：并行度，默认1；
initializationMode：初始中心算法，默认"k-means||"；initializationSteps：初始步长，默
认5；epsilon：中心距离阈值，默认1e-4；seed：随机种子。
  /**
   * Constructs a KMeans instance with default parameters: {k: 2,
maxIterations: 20, runs: 1,
   * initializationMode: "k-means||", initializationSteps: 5, epsilon: 1e-4,
seed: random}.
   */
  defthis() =this(2,20, 1, KMeans.K_MEANS_PARALLEL,5, 1e-4,
Utils.random.nextLong())
//参数设置
/** Set the number of clusters to create (k). Default: 2. */
  def setK(k: Int):this.type = {
    this.k = k
    this
  }
**省略各个参数设置代码**
// run 方法，KMeans 主入口函数
  /**
```

```
   * Train a K-means model on the given set of points; `data` should be cached
for high
   * performance, because this is an iterative algorithm.
   */
  def run(data: RDD[Vector]): KMeansModel = {

    if (data.getStorageLevel == StorageLevel.NONE) {
      logWarning("The input data is not directly cached, which may hurt
performance if its"
        + " parent RDDs are also uncached.")
    }

// Compute squared norms and cache them.
// 计算每行数据的 L2范数，数据转换：data[Vector]=> data[(Vector, norms)]，其中 norms 是
Vector 的 L2范数，norms 就是：
    val norms = data.map(Vectors.norm(_,2.0))
    norms.persist()
    val zippedData = data.zip(norms).map {case (v, norm) =>
      new VectorWithNorm(v, norm)
    }
    val model = runAlgorithm(zippedData)
    norms.unpersist()

    // Warn at the end of the run as well, for increased visibility.
    if (data.getStorageLevel == StorageLevel.NONE) {
      logWarning("The input data was not directly cached, which may hurt
performance if its"
        + " parent RDDs are also uncached.")
    }
    model
  }
// runAlgorithm 方法，KMeans 实现方法。
  /**
   * Implementation of K-Means algorithm.
   */
  privatedef runAlgorithm(data: RDD[VectorWithNorm]): KMeansModel = {

    val sc = data.sparkContext
```

```
    val initStartTime = System.nanoTime()

    val centers =if (initializationMode == KMeans.RANDOM) {
      initRandom(data)
    } else {
      initKMeansParallel(data)
    }

    val initTimeInSeconds = (System.nanoTime() - initStartTime) /1e9
    logInfo(s"Initialization with $initializationMode took
" +"%.3f".format(initTimeInSeconds) +
      " seconds.")

    val active = Array.fill(runs)(true)
    val costs = Array.fill(runs)(0.0)

    var activeRuns =new ArrayBuffer[Int] ++ (0 until runs)
    var iteration =0

    val iterationStartTime = System.nanoTime()
//KMeans 迭代执行，计算每个样本属于哪个中心点，中心点累加样本的值及计数，然后根据中心点的所有的样本数据进行中心点的更新，并比较更新前的数值，判断是否完成。其中 runs 代表并行度。
// Execute iterations of Lloyd's algorithm until all runs have converged
    while (iteration < maxIterations && !activeRuns.isEmpty) {
      type WeightedPoint = (Vector, Long)
      def mergeContribs(x: WeightedPoint, y: WeightedPoint): WeightedPoint = {
        axpy(1.0, x._1, y._1)
        (y._1, x._2 + y._2)
      }

      val activeCenters = activeRuns.map(r => centers(r)).toArray
      val costAccums = activeRuns.map(_ => sc.accumulator(0.0))

      val bcActiveCenters = sc.broadcast(activeCenters)

// Find the sum and count of points mapping to each center
//计算属于每个中心点的样本，对每个中心点的样本进行累加和计算;
runs 代表并行度，k 中心点个数，sums 代表中心点样本累加值，counts 代表中心点样本计数;
contribs 代表（（并行度 I，中心 J），（中心 J 样本之和，中心 J 样本计数和））;
```

```
findClosest 方法：找到点与所有聚类中心最近的一个中心；
      val totalContribs = data.mapPartitions { points =>
        val thisActiveCenters = bcActiveCenters.value
        val runs = thisActiveCenters.length
        val k = thisActiveCenters(0).length
        val dims = thisActiveCenters(0)(0).vector.size

        val sums = Array.fill(runs, k)(Vectors.zeros(dims))
        val counts = Array.fill(runs, k)(0L)

        points.foreach { point =>
          (0 until runs).foreach { i =>
           val (bestCenter, cost) = KMeans.findClosest(thisActiveCenters(i),
point)
           costAccums(i) += cost
           val sum = sums(i)(bestCenter)
           axpy(1.0, point.vector, sum)
           counts(i)(bestCenter) += 1
          }
        }

        val contribs =for (i <-0 until runs; j <-0 until k) yield {
          ((i, j), (sums(i)(j), counts(i)(j)))
        }
        contribs.iterator
      }.reduceByKey(mergeContribs).collectAsMap()
//更新中心点，更新中心点= sum/count；
判断 newCenter 与 centers 之间的距离是否 > epsilon * epsilon;
// Update the cluster centers and costs for each active run
      for ((run, i) <- activeRuns.zipWithIndex) {
        var changed =false
        var j =0
        while (j < k) {
          val (sum, count) = totalContribs((i, j))
          if (count !=0) {
           scal(1.0 / count, sum)
           val newCenter =new VectorWithNorm(sum)
           if (KMeans.fastSquaredDistance(newCenter, centers(run)(j)) >
epsilon * epsilon) {
```

```
             changed = true
           }
           centers(run)(j) = newCenter
          }
          j += 1
        }
        if (!changed) {
          active(run) = false
          logInfo("Run " + run +" finished in " + (iteration +1) + "
iterations")
        }
        costs(run) = costAccums(i).value
      }

      activeRuns = activeRuns.filter(active(_))
      iteration += 1
    }

    val iterationTimeInSeconds = (System.nanoTime() - iterationStartTime) /1e9
    logInfo(s"Iterations took " +"%.3f".format(iterationTimeInSeconds) +"
seconds.")

    if (iteration == maxIterations) {
      logInfo(s"KMeans reached the max number of iterations: $maxIterations.")
    } else {
      logInfo(s"KMeans converged in $iteration iterations.")
    }

    val (minCost, bestRun) = costs.zipWithIndex.min

    logInfo(s"The cost for the best run is $minCost.")

    new KMeansModel(centers(bestRun).map(_.vector))
  }
//findClosest 方法：找到点与所有聚类中心最近的一个中心；
/**
   * Returns the index of the closest center to the given point, as well as
the squared distance.
   */
```

```
  private[mllib]def findClosest(
      centers: TraversableOnce[VectorWithNorm],
      point: VectorWithNorm): (Int, Double) = {
    var bestDistance = Double.PositiveInfinity
    var bestIndex =0
    var i =0
    centers.foreach { center =>
 // Since '\|a - b\| \geq |\|a\| - \|b\||', we can use this lower bound to
avoid unnecessary
 // distance computation.
      var lowerBoundOfSqDist = center.norm - point.norm
      lowerBoundOfSqDist = lowerBoundOfSqDist * lowerBoundOfSqDist
      if (lowerBoundOfSqDist < bestDistance) {
        val distance: Double = fastSquaredDistance(center, point)
        if (distance < bestDistance) {
          bestDistance = distance
          bestIndex = i
        }
      }
      i += 1
    }
    (bestIndex, bestDistance)
  }
```

在上面 findClosest 方法中有两行黑体字代码，如果中心点 center 是(a_1,b_1)，需要计算的点 point 是(a_2,b_2)，那么 lowerBoundOfS-qDist 是：

$$(\sqrt{a_1^2+b_1^2}-\sqrt{a_2^2+b_2^2})^2 \tag{5-27}$$

如下是展开式，第二个是真正计算欧式距离时的除去开平方的公式。在查找最短距离的时候无须计算开方，因为只需要计算出开方里面的式子就可以进行比较了，MLlib 也是这样做的。

$$\begin{aligned}(\sqrt{a_1^2+b_1^2}-\sqrt{a_2^2+b_2^2})^2 &= a_1^2+b_1^2+a_2^2+b_2^2-2\sqrt{(a_1^2+b_1^2)(a_2^2+b_2^2)} \\ (a_1-a_2)^2+(b_1-b_2)^2 &= a_1^2+b_1^2+a_2^2+b_2^2-2(a_1a_2+b_1b_2)\end{aligned} \tag{5-28}$$

在进行距离比较的时候，先计算很容易计算的 lowerBoundOfSqDist，如果 lowerBoundOfSqDist 都不小于之前计算得到的最小距离 bestDistance，那真正的欧式距离也不可能小于 bestDistance 了，因此这种情况下就不需要去计算欧式距离，省去很多计算工作。

如果 lowerBoundOfSqDist 小于 bestDistance，则进行距离的计算，调用

fastSquaredDistance，这个方法将调用 MLUtils.Scala 里面的 fastSquaredDistance 方法，计算真正的欧式距离，代码如下：

```
/**
   * Returns the squared Euclidean distance between two vectors. The following
formula will be used
   * if it does not introduce too much numerical error:
   * <pre>
   *   \|a - b\|_2^2 = \|a\|_2^2 + \|b\|_2^2 - 2 a^T b.
   * </pre>
   * When both vector norms are given, this is faster than computing the
squared distance directly,
   * especially when one of the vectors is a sparse vector.
   *
   * @param v1 the first vector
   * @param norm1 the norm of the first vector, non-negative
   * @param v2 the second vector
   * @param norm2 the norm of the second vector, non-negative
   * @param precision desired relative precision for the squared distance
   * @return squared distance between v1 and v2 within the specified precision
   */
  private[mllib]def fastSquaredDistance(
      v1: Vector,
      norm1: Double,
      v2: Vector,
      norm2: Double,
      precision: Double = 1e-6): Double = {
    val n = v1.size
    require(v2.size == n)
    require(norm1 >= 0.0 && norm2 >=0.0)
    val sumSquaredNorm = norm1 * norm1 + norm2 * norm2
    val normDiff = norm1 - norm2
    var sqDist =0.0
    /*
     * The relative error is
     * <pre>
     * EPSILON * ( \|a\|_2^2 + \|b\\_2^2 + 2 |a^T b|) / ( \|a - b\|_2^2 ),
     * </pre>
     * which is bounded by
```

```
     * <pre>
     * 2.0 * EPSILON * ( \|a\|_2^2 + \|b\|_2^2 ) / ( (\|a\|_2 - \|b\|_2)^2 ).
     * </pre>
     * The bound doesn't need the inner product, so we can use it as a
sufficient condition to
     * check quickly whether the inner product approach is accurate.
     */
    val precisionBound1 =2.0 * EPSILON * sumSquaredNorm / (normDiff * normDiff
+ EPSILON)
    if (precisionBound1 < precision) {
      sqDist = sumSquaredNorm - 2.0 * dot(v1, v2)
    } elseif (v1.isInstanceOf[SparseVector] || v2.isInstanceOf[SparseVector])
{
      val dotValue = dot(v1, v2)
      sqDist = math.max(sumSquaredNorm - 2.0 * dotValue,0.0)
      val precisionBound2 = EPSILON * (sumSquaredNorm +2.0 *
math.abs(dotValue)) /
        (sqDist + EPSILON)
      if (precisionBound2 > precision) {
        sqDist = Vectors.sqdist(v1, v2)
      }
    } else {
      sqDist = Vectors.sqdist(v1, v2)
    }
    sqDist
  }
```

fastSquaredDistance 方法会先计算一个精度，有关精度的计算 val precisionBound1 = 2.0 * EPSILON * sumSquaredNorm / (normDiff * normDiff + EPSILON)，如果在精度满足条件的情况下，欧式距离 sqDist = sumSquaredNorm - 2.0 * v1.dot(v2)，sumSquaredNorm 即为 $a_1^2+b_1^2+a_2^2+b_2^2$，2.0 * v1.dot(v2)即为 $2(a_1a_2+b_1b_2)$。这也是之前将 norm 计算出来的好处。如果精度不满足要求，则进行原始的距离计算公式 $(a_1-a_2)^2+(b_1-b_2)^2$，即调用 Vectors.sqdist(v1, v2)。

5.7.3　MLlib KMeans 实例

1. 数据

数据格式为：特征 1 特征 2 特征 3。

```
0.0 0.0 0.0
```

```
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

2. 代码

```
//1读取样本数据
  valdata_path ="/home/jb-huangmeiling/kmeans_data.txt"
  valdata =sc.textFile(data_path)
  valexamples =data.map { line => Vectors.dense(line.split('
').map(_.toDouble))
  }.cache()
  valnumExamples =examples.count()
  println(s"numExamples = $numExamples.")
//2建立模型
  valk =2
  valmaxIterations =20
  valruns =2
  valinitializationMode ="k-means||"
  valmodel = KMeans.train(examples,k, maxIterations,runs, initializationMode)
//3计算测试误差
  valcost =model.computeCost(examples)
  println(s"Total cost = $cost.")
```

5.8 Spark MLlib FPGrowth 关联规则算法

5.8.1 基本概念

关联规则挖掘的一个典型例子是购物篮分析。关联规则研究有助于发现交易数据库中不同商品（项）之间的联系，找出顾客购买行为模式，如购买了某一商品对购买其他商品的影响，分析结果可以应用于商品货架布局、货存安排以及根据购买模式对用户进行分类。

关联规则的相关术语如下：

（1）项与项集

这是一个集合的概念，在一篮子商品中的一件消费品即为一项（Item），则若干项的集合为项集，如{啤酒，尿布}构成一个二元项集。

（2）关联规则

一般表示的形式，X 为先决条件，Y 为相应的关联结果，用于表示数据内隐含的关联性。如：表示购买了尿布的消费者往往也会购买啤酒。关联性强度如何，由三个概念——支持度、置信度、提升度来控制和评价。例：有 10000 个消费者购买了商品，其中购买尿布 1000 个，购买啤酒 2000 个，购买面包 500 个，同时购买尿布和啤酒 800 个，同时购买尿布和面包 100 个。

（3）支持度（Support）

支持度是指在所有项集中{X, Y}出现的可能性，即项集中同时含有 X 和 Y 的概率。该指标作为建立强关联规则的第一个门槛，衡量了所考察关联规则在“量”上的多少。通过设定最小阈值（minsup），剔除“出镜率”较低的无意义规则，保留出现较为频繁的项集所隐含的规则。

设定最小阈值为 5%，由于{尿布，啤酒}的支持度为 800/10000=8%，满足基本输入要求，成为频繁项集，保留规则；而{尿布，面包}的支持度为 100/10000=1%，被剔除。

（4）置信度（Confidence）

置信度表示在先决条件 X 发生的条件下，关联结果 Y 发生的概率。这是生成强关联规则的第二个门槛，衡量了所考察的关联规则在“质”上的可靠性。相似的，我们需要对置信度设定最小阈值（mincon）来实现进一步筛选。具体的，当设定置信度的最小阈值为 70%时，置信度为 800/1000=80%，而置信度为 800/2000=40%，被剔除。

（5）提升度（lift）

提升度表示在含有 X 的条件下同时含有 Y 的可能性与没有 X 这个条件下项集中含有 Y 的可能性之比，公式为 confidence(artichok => cracker)/support(cracker) = 80%/50% = 1.6。该指标与置信度同样衡量规则的可靠性，可以看作是置信度的一种互补指标。

5.8.2　FPGrowth 算法

FP-Growth（频繁模式增长）算法是韩家炜老师在 2000 年提出的关联分析算法，它采取如下分治策略：将提供频繁项集的数据库压缩到一棵频繁模式树（FP-Tree），但仍保留项集关联信息。该算法和 Apriori 算法最大的不同有两点：第一，不产生候选集；第二，只需要两次遍历数据库，大大提高了效率。

1. 构造 FP-树

（1）扫描事务数据库 D 一次。收集频繁项的集合 F 和它们的支持度。对 F 按支持度降序排序，结果为频繁项表 L。

（2）创建 FP-树的根结点，以“null”标记它。对于 D 中每个事务 Trans，执行下面操作：选择 Trans 中的频繁项，并按 L 中的次序排序。设排序后的频繁项表为[p | P]，其中，p 是第一个元素，而 P 是剩余元素的表。调用 insert_tree([p | P], T)。该过程执行情况如下：如果 T 有子女 N，使得 N.item-name = p.item-name，则 N 的计数增加 1；否则创建一个新节点 N，将其计数设置为 1，链接到它的父节点 T，并且通过节点链结将其链接到具有相同 item-

name 的节点。如果 *P* 非空，递归地调用 insert_tree(*P*, *N*)。

2. FP-树的挖掘

通过调用 FP_growth(FP_tree, null)实现。该过程实现如下：

```
FP_growth(Tree, α)
{
  if Tree 含单个路径 P then
    for 路径 P 中结点的每个组合（记作 β
      产生模式 β∪α，其支持度 support = β 中结点的最小支持度
    else for each ai 在 Tree 的头部(按照支持度由低到高顺序进行扫描) {
      产生一个模式 β = ai∪α，其支持度 support = ai.support
      构造 β 的条件模式基，然后构造 β 的条件 FP-树 Treeβ
      if Treeβ ≠ ∅ then
        调用 FP_growth (Treeβ, β);
    }
  end
}
```

3. FP-Growth 算法构造 FP-树

（1）事务数据库建立

原始事务数据库如表 5-12 所示。

表 5-12　原始事务数据库

Tid	Items
1	I1,I2,I5
2	I2,I4
3	I2,I3
4	I1,I2,I4
5	I1,I3
6	I2,I3
7	I1,I3
8	I1,I2,I3,I5
9	I1,I2,I3

扫描事务数据库得到频繁 1-项目集 F：

I1	I2	I3	I4	I5
6	7	6	2	2

定义 minsup=20%，即最小支持度为 2，重新排列 F：

I2	I1	I3	I4	I5
7	6	6	2	2

这样，重新调整事务数据库如表 5-13 所示。

表 5-13　重新调整事务数据库

Tid	Items
1	I2, I1,I5
2	I2,I4
3	I2,I3
4	I2, I1,I4
5	I1,I3
6	I2,I3
7	I1,I3
8	I2, I1,I3,I5
9	I2, I1,I3

（2）创建根节点和频繁项目表，如图 5-6 所示。

Item-name	Node-head
I2	Null
I1	Null
I3	Null
I4	Null
I5	Null

Null

图 5-6　创建根节点和频繁项目表

（3）加入第一个事务(I2,I1,I5)，如图 5-7 所示。

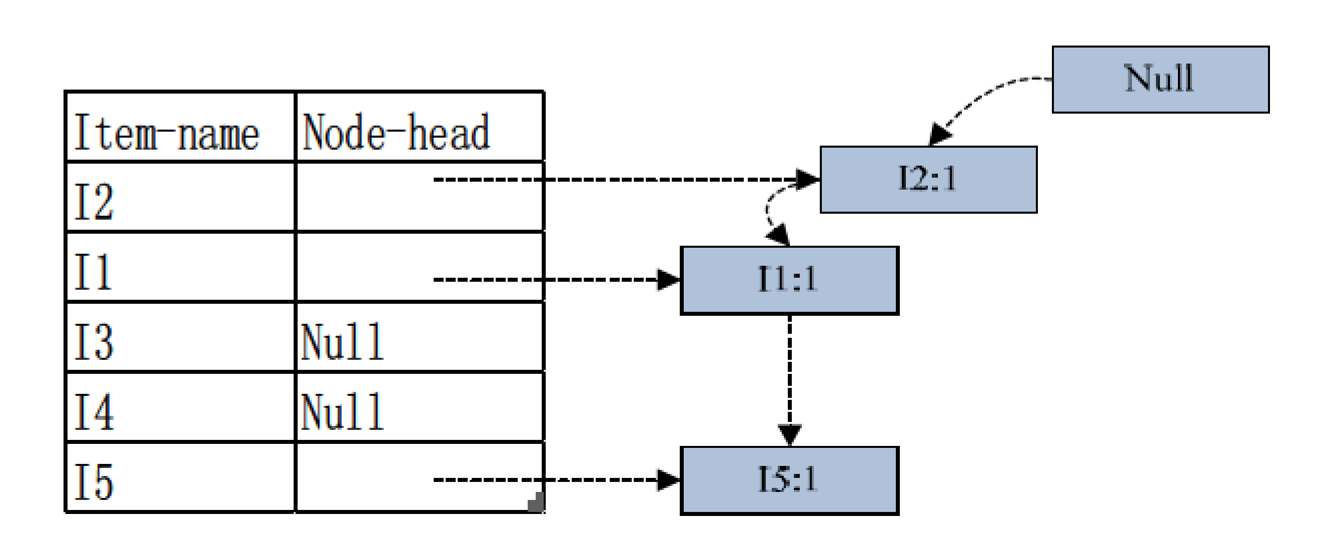


图 5-7　加入第一个事务(I2,I1,I5)

（4）加入第二个事务(I2,I4)，如图 5-8 所示。

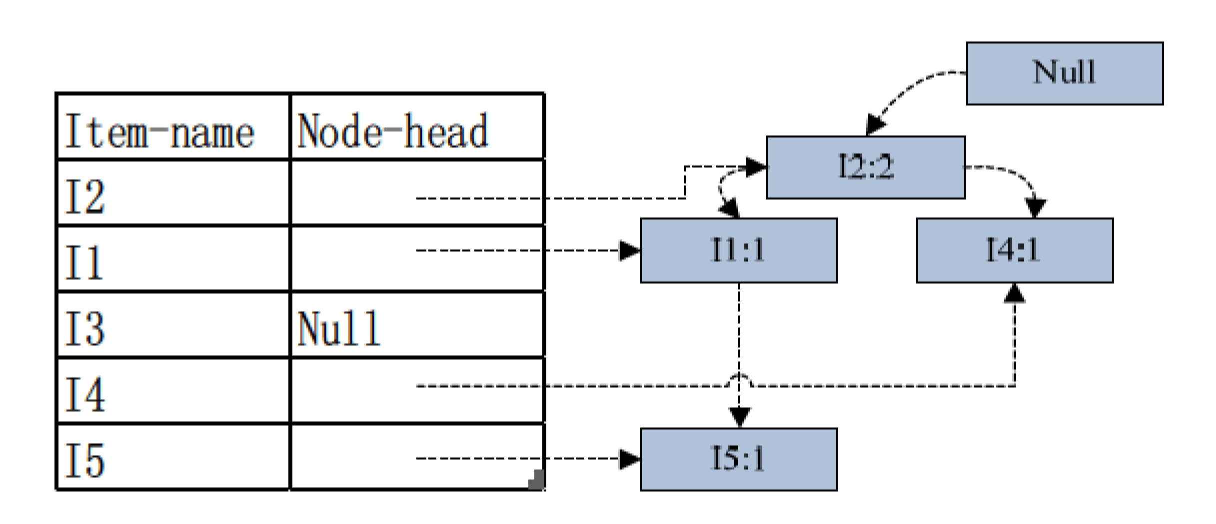


图 5-8　加入第二个事务(I2,I4)

（5）加入第三个事务(I2,I3)，如图 5-9 所示。

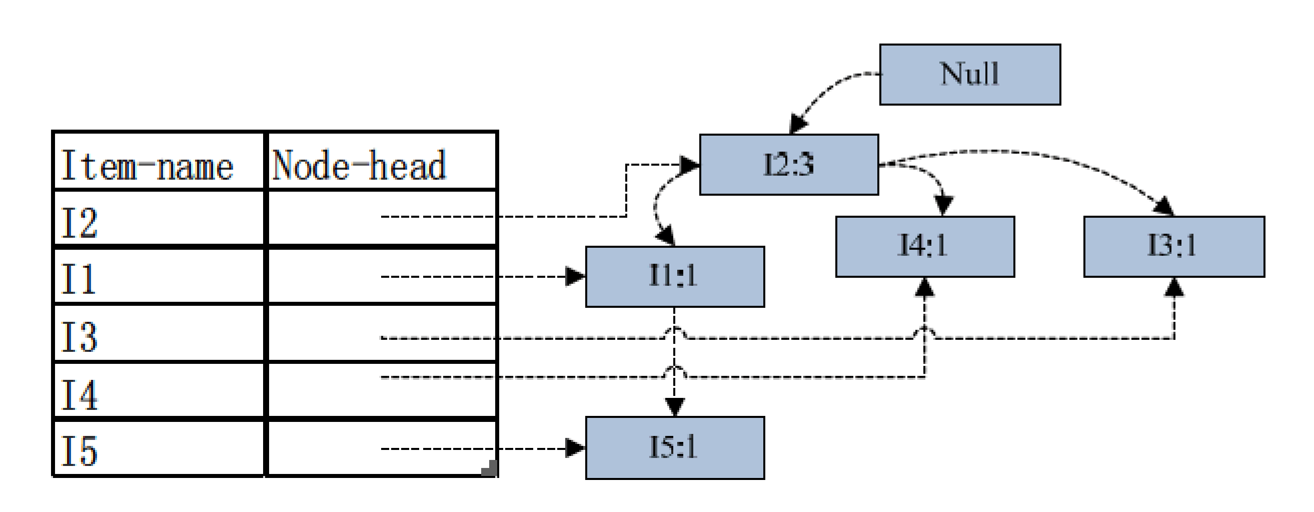


图 5-9　加入第三个事务(I2,I3)

以此类推加入第 5、6、7、8、9 个事务。

（6）加入第 9 个事务(I2,I1,I3)，如图 5-10 所示。

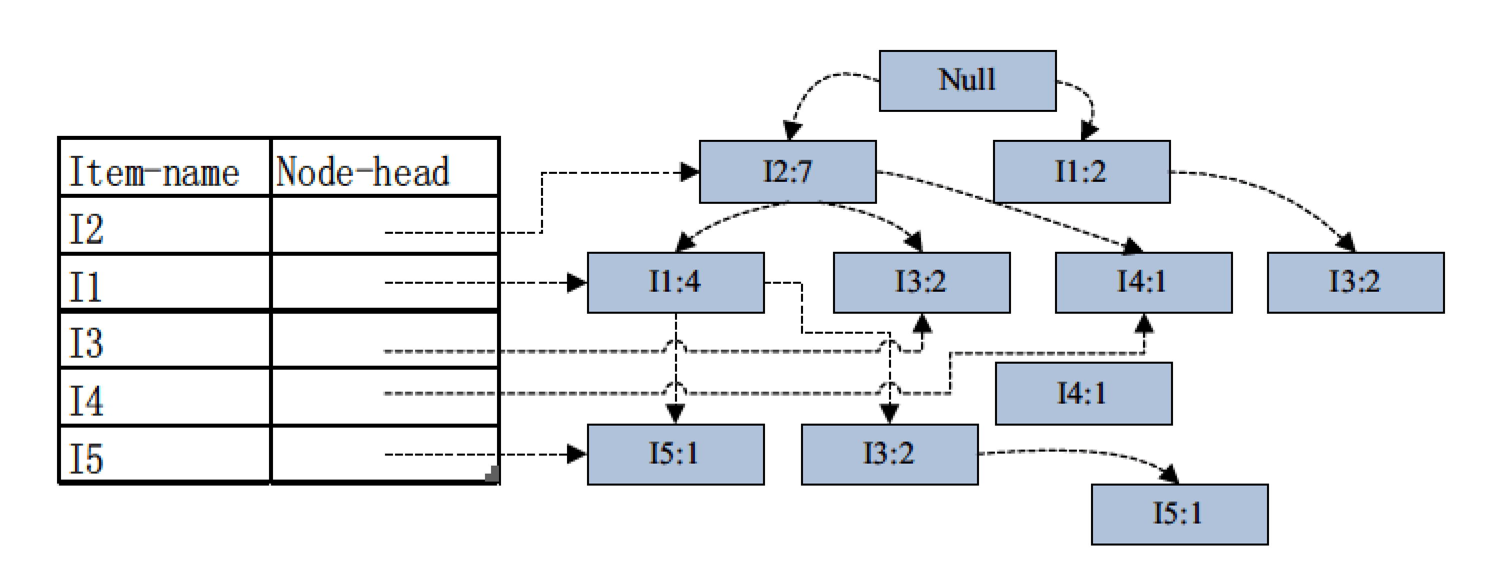


图 5-10　加入第 9 个事务(I2,I1,I3)

4. FP-Growth 算法 FP-树挖掘

FP-树建好后，就可以进行频繁项集的挖掘，挖掘算法称为 FpGrowth（Frequent Pattern Growth）算法，挖掘从表头 header 的最后一个项开始，以此类推。本文以 I5 为例进行挖掘。

对于 I5，得到条件模式基：<(I2,I1:1)>、<I2,I1,I3:1>，构造条件 FP-tree，如图 5-11 所示。

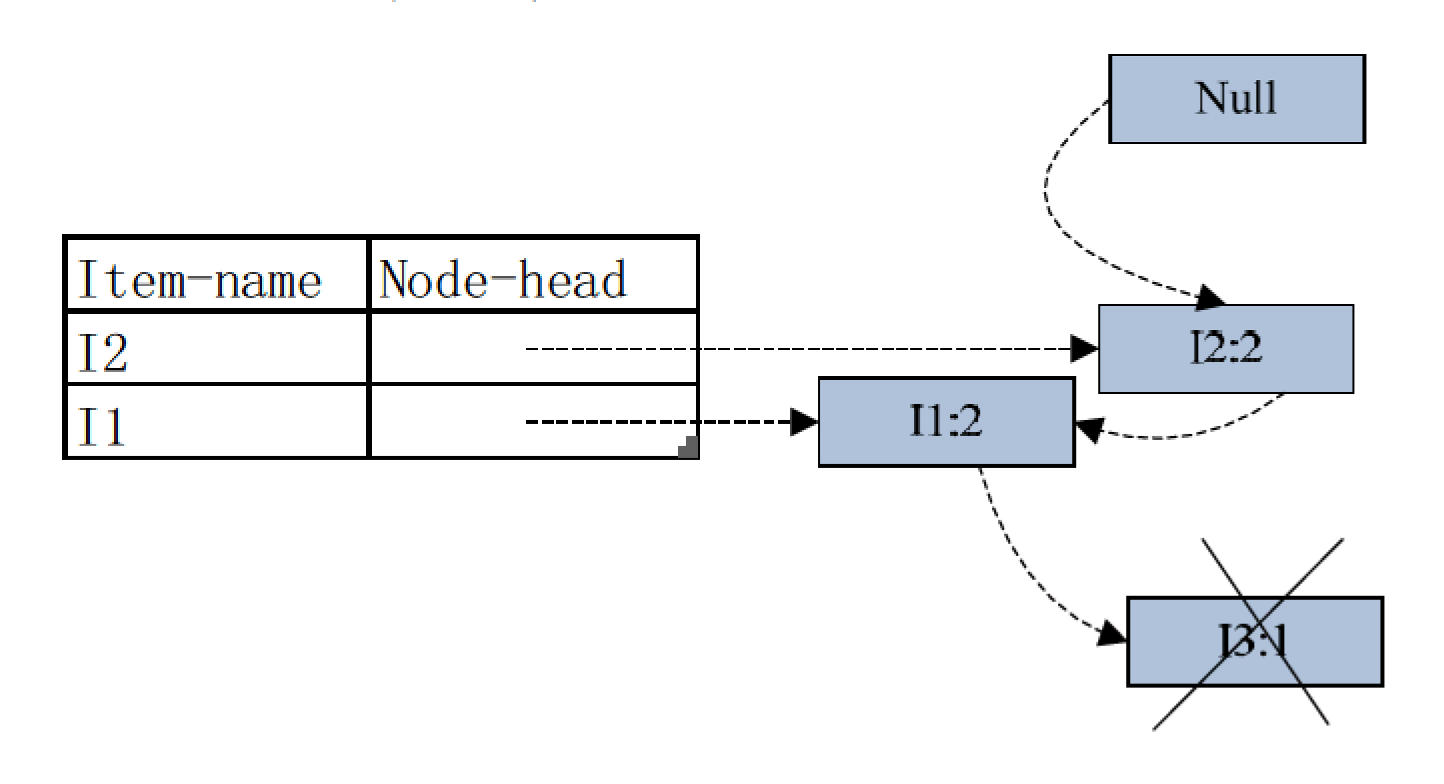


图 5-11　构造条件 FP-tree

得到 I5 频繁项集：{{I2,I5:2},{I1,I5:2},{I2,I1,I5:2}}，I4、I1 的挖掘与 I5 类似，条件 FP-树都是单路径。

5.8.3 Spark MLlib FPGrowth 源码分析

FPGrowth 源码包括：FPGrowth、FPTree 两部分。

- FPGrowth 中包括：run 方法、genFreqItems 方法、genFreqItemsets 方法、genCondTransactions 方法。
- FPTree 中包括：add 方法、merge 方法、project 方法、getTransactions 方法、extract 方法。

```
// run 计算频繁项集
  /**
   * Computes an FP-Growth model that contains frequent itemsets.
   * @param data input data set, each element contains a transaction
   * @return an [[FPGrowthModel]]
   */
  def run[Item: ClassTag](data: RDD[Array[Item]]): FPGrowthModel[Item] = {
    if (data.getStorageLevel == StorageLevel.NONE) {
      logWarning("Input data is not cached.")
    }
    val count = data.count()//计算事务总数
    val minCount = math.ceil(minSupport * count).toLong//计算最小支持度
    val numParts = if (numPartitions > 0)
numPartitions else data.partitions.length
val partitioner = new HashPartitioner(numParts)
//freqItems 计算满足最小支持度的 Items 项
val freqItems = genFreqItems(data, minCount, partitioner)
//freqItemsets 计算频繁项集
    val freqItemsets = genFreqItemsets(data, minCount, freqItems, partitioner)
    new FPGrowthModel(freqItemsets)
}
// genFreqItems 计算满足最小支持度的 Items 项
/**
   * Generates frequent items by filtering the input data using minimal
support level.
   * @param minCount minimum count for frequent itemsets
   * @param partitioner partitioner used to distribute items
   * @return array of frequent pattern ordered by their frequencies
   */
  privatedef genFreqItems[Item: ClassTag](
      data: RDD[Array[Item]],
```

```
      minCount: Long,
      partitioner: Partitioner): Array[Item] = {
    data.flatMap { t =>
      val uniq = t.toSet
      if (t.size != uniq.size) {
        thrownew SparkException(s"Items in a transaction must be unique but
got ${t.toSeq}.")
      }
      t
    }.map(v => (v, 1L))
      .reduceByKey(partitioner, _ + _)
      .filter(_._2 >= minCount)
      .collect()
      .sortBy(-_._2)
      .map(_._1)
}//统计每个 Items 项的频次，对小于 minCount 的 Items 项过滤，返回 Items 项。
 // genFreqItemsets 计算频繁项集：生成 FP-Trees，挖掘 FP-Trees
/**
   * Generate frequent itemsets by building FP-Trees, the extraction is done
on each partition.
   * @param data transactions
   * @param minCount minimum count for frequent itemsets
   * @param freqItems frequent items
   * @param partitioner partitioner used to distribute transactions
   * @return an RDD of (frequent itemset, count)
   */
  privatedef genFreqItemsets[Item: ClassTag](
      data: RDD[Array[Item]],
      minCount: Long,
      freqItems: Array[Item],
      partitioner: Partitioner): RDD[FreqItemset[Item]] = {
    val itemToRank = freqItems.zipWithIndex.toMap//表头
    data.flatMap { transaction =>
      genCondTransactions(transaction, itemToRank, partitioner)
    }.aggregateByKey(new FPTree[Int], partitioner.numPartitions)( //生成 FP 树
      (tree, transaction) => tree.add(transaction, 1L),//FP 树增加一条事务
      (tree1, tree2) => tree1.merge(tree2)) //FP 树合并
    .flatMap { case (part, tree) =>
      tree.extract(minCount, x => partitioner.getPartition(x) == part)//FP 树挖
```

```
掘频繁项
    }.map { case (ranks, count) =>
      new FreqItemset(ranks.map(i => freqItems(i)).toArray, count)
    }
}
// add FP-Trees 增加一条事务数据
/** Adds a transaction with count. */
  def add(t: Iterable[T], count: Long = 1L): this.type = {
    require(count > 0)
    var curr = root
    curr.count += count
    t.foreach { item =>
      val summary = summaries.getOrElseUpdate(item, new Summary)
      summary.count += count
      val child = curr.children.getOrElseUpdate(item, {
        val newNode = new Node(curr)
        newNode.item = item
        summary.nodes += newNode
        newNode
      })
      child.count += count
      curr = child
    }
    this
}
// merge FP-Trees 合并
  /** Merges another FP-Tree. */
  def merge(other: FPTree[T]): this.type = {
    other.transactions.foreach { case (t, c) =>
      add(t, c)
    }
    this
}
// extract FP-Trees 挖掘，返回所有频繁项集
  /** Extracts all patterns with valid suffix and minimum count. */
  def extract(
      minCount: Long,
      validateSuffix: T => Boolean = _ => true): Iterator[(List[T], Long)] = {
    summaries.iterator.flatMap { case (item, summary) =>
```

```
      if (validateSuffix(item) && summary.count >= minCount) {
        Iterator.single((item :: Nil, summary.count)) ++
          project(item).extract(minCount).map { case (t, c) =>
            (item :: t, c)
          }
      } else {
        Iterator.empty
      }
    }
  }
}
```

5.9 Spark MLlib 协同过滤推荐算法

5.9.1 协同过滤概念

协同过滤常常被用于分辨某位特定顾客可能感兴趣的东西，这些结论来自于对其他相似顾客对哪些产品感兴趣的分析。协同过滤以其出色的速度和健壮性，在全球互联网领域炙手可热。

协同过滤推荐（Collaborative Filtering recommendation）是在信息过滤和信息系统中正迅速成为一项很受欢迎的技术。与传统的基于内容过滤直接分析内容进行推荐不同，协同过滤分析用户兴趣，在用户群中找到指定用户的相似用户，综合这些相似用户对某一信息的评价，形成系统对该指定用户对此信息的喜好程度预测[87-89]。与传统文本过滤相比，协同过滤有下列优点:

- 能够过滤难以进行机器自动基于内容分析的信息，如艺术品、音乐。
- 能够基于一些复杂的，难以表达的概念（信息质量、品位）进行过滤。
- 推荐的新颖性。

正因为如此，协同过滤在商业应用上也取得了不错的成绩。Amazon、CDNow、MovieFinder 都采用了协同过滤的技术来提高服务质量。也存在缺点:

- 用户对商品的评价非常稀疏，这样基于用户的评价所得到的用户间的相似性可能不准确（即稀疏性问题）。
- 随着用户和商品的增多，系统的性能会越来越低。
- 如果从来没有用户对某一商品加以评价，则这个商品就不可能被推荐（即最初评价问题）。

通常，协同过滤算法按照数据使用，可以分为：

- 基于用户（UserCF）。
- 基于商品（ItemCF）。
- 基于模型（ModelCF）。

按照模型，可以分为：

- 最近邻模型：基于距离的协同过滤算法。
- Latent Factor Mode（SVD）：基于矩阵分解的模型。
- Graph：图模型，社会网络图模型。

5.9.2 相似度度量

关于相似度的计算，现有的几种基本方法都是基于向量（Vector）的，其实也就是计算两个向量的距离，距离越近，相似度越大。在推荐的场景中，在用户-物品偏好的二维矩阵中，我们可以将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，或者将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度。下面我们详细介绍几种常用的相似度计算方法。

1. 欧几里得距离（Euclidean Distance）

最初用于计算欧几里得空间中两个点的距离，假设 x，y 是 n 维空间的两个点，它们之间的欧几里得距离是：

$$d(x,y)=\sqrt{\left(\sum_{i}^{n}(x_i-y_i)^2\right)} \qquad (5\text{-}29)$$

相似度：

$$sim(x,y)=\frac{1}{1+d(x,y)} \qquad (5\text{-}30)$$

2. 皮尔逊相关度（Pearson Correlation Coefficient）

皮尔逊相关度（Pearson Correlation Coefficient），用于判断两组数据与某一直线拟合程度的一种度量，取值在[-1,1]之间。当数据不是很规范的时候（如偏差较大），皮尔逊相关度会给出较好的结果。

$$p(x,y)=\frac{\sum x_i y_i - n\overline{xy}}{(n-1)s_x s_y}=\frac{n\sum x_i y_i-\sum x_i\sum y_i}{\sqrt{n\sum x_j^2-(\sum x_j)^2}\sqrt{n\sum y_j^2-(\sum y_j)^2}} \qquad (5\text{-}31)$$

3. 曼哈顿距离

曼哈顿距离（Manhattan distance），就是在欧几里得空间的固定直角坐标系上两点所形成的线段对轴产生的投影的距离总和：

$$d(x,y)=\sum // x_i - y_i // \tag{5-32}$$

4. Jaccard 系数

Jaccard 系数，也称为 Tanimoto 系数，是 Cosine 相似度的扩展，也多用于计算文档数据的相似度。通常应用于 x 为布尔向量，即各分量只取 0 或 1 的时候。此时，表示的是 x，y 的公共特征的占 x，y 所占有的特征的比例:

$$T(x,y)=\frac{x \cdot y}{//x//^2 + //y//^2 - x \cdot y}=\frac{\sum x_i y_i}{\sqrt{\sum x_j^2}+\sqrt{\sum y_j^2}-\sum x_i y_i} \tag{5-33}$$

5.9.3 协同过滤算法按照数据使用分类

1. 基于用户（UserCF）——基于用户相似性

基于用户的协同过滤，通过不同用户对物品的评分来评测用户之间的相似性，基于用户之间的相似性做出推荐。简单来讲，就是给用户推荐和他兴趣相似的其他用户喜欢的物品。

举个例子：有三个用户 A、B、C，四个物品 A、B、C、D，需要向用户 A 推荐物品，如表 5-14 所示。

表 5-14 向用户 A 推荐物品表

用户/物品	物品 A	物品 B	物品 C	物品 D
用户 A	√		√	推荐
用户 B		√		
用户 C	√		√	√

这里，由于用户 A 和用户 C 都买过物品 A 和物品 C，所以，我们认为用户 A 和用户 C 非常相似，同时，用户 C 又买过物品 D，那么就需要给 A 用户推荐物品 D。

基于 UserCF 的基本思想相当简单，基于用户对物品的偏好，找到相邻邻居用户，然后将邻居用户喜欢的商品推荐给当前用户。

计算上，将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，找到 K 邻居后，根据邻居的相似度权重以及他们对物品的偏好，预测当前用户没有偏好的未涉及物品，计算得到一个排序的物品列表作为推荐。

2. 基于商品（ItemCF）——基于商品相似性

基于商品的协同过滤，通过用户对不同 Item 的评分来评测 Item 之间的相似性，基于 Item 之间的相似性做出推荐。简单来讲，就是给用户推荐和他之前喜欢的物品相似的物品。

有三个用户 A、B、C 和三件物品 A、B、C，需要向用户 C 推荐物品。这里，由于用户 A 买过物品 A 和 C，用户 B 买过物品 A、B、C，用户 C 买过物品 A，从用户 A 和 B 可以看出，这两个用户都买过物品 A 和 C，说明物品 A 和 C 非常相似，同时，用户 C 又买过物品 A，所以，将物品 C 推荐给用户 C，如表 5-15 所示。

表 5-15　将物品 C 推荐给用户 C 表

用户/物品	物品 A	物品 B	物品 C
用户 A	√		√
用户 B	√	√	√
用户 C	√		推荐

基于 ItemCF 的原理和基于 UserCF 类似，只是在计算邻居时采用物品本身，而不是从用户的角度，即基于用户对物品的偏好找到相似的物品，然后根据用户的历史偏好，推荐相似的物品给他。

从计算角度，即将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品后，根据用户历史的偏好预测当前用户还没有表示偏好的物品，计算得到一个排序的物品列表作为推荐。

3. 基于模型（ModelCF）

基于模型的协同过滤推荐就是基于样本的用户喜好信息，训练一个推荐模型，然后根据实时的用户喜好的信息进行预测，计算推荐。

5.9.4　Spark MLlib 协同过滤算法实现

Spark MLlib 实现了交替最小二乘法（ALS）来学习这些隐性语义因子。在 MLlib 中的实现有如下的参数：

- numBlocks：用于并行化计算的分块个数 (设置为-1，为自动配置)。
- rank：模型中隐语义因子的个数。
- iterations：迭代的次数。
- lambda：ALS 的正则化参数。
- implicitPrefs：决定了是用显性反馈 ALS 的版本还是用隐性反馈数据集的版本。
- alpha：是一个针对隐性反馈 ALS 版本的参数，这个参数决定了偏好行为强度的基准。

可以调整这些参数，不断优化结果，使均方差变小。比如：iterations 越多，lambda 较小，均方差会较小，推荐结果较优。

协同过滤 ALS 算法推荐过程如下：

- 加载数据到 ratings RDD，每行记录包括：user、product、rate。
- 从 ratings 得到用户商品的数据集：(user, product)。
- 使用 ALS 对 ratings 进行训练。
- 通过 model 对用户商品进行预测。评分：((user, product), rate)。
- 从 ratings 得到用户商品的实际评分：((user, product), rate)。
- 合并预测评分和实际评分的两个数据集，并求均方差。

在下面的例子中我们额定载荷数据。每一行包含一个用户、一个产品和评级。 我们使用 train() ALS 方法，采用明确的额定值。我们评估的推荐模型，通过测量的均方根误差等级预测。

1. Scala

ALS Scala docs 更多的细节参考 API。

```
importorg.apache.spark.mllib.recommendation.ALS
importorg.apache.spark.mllib.recommendation.MatrixFactorizationModel
importorg.apache.spark.mllib.recommendation.Rating

// Load and parse the data
valdata=sc.textFile("data/mllib/als/test.data")
valratings=data.map(_.split(',')match{caseArray(user,item,rate)=>
Rating(user.toInt,item.toInt,rate.toDouble)
})

// Build the recommendation model using ALS
valrank=10
valnumIterations=10
valmodel=ALS.train(ratings,rank,numIterations,0.01)

// Evaluate the model on rating data
valusersProducts=ratings.map{caseRating(user,product,rate)=>
(user,product)
}
valpredictions=
model.predict(usersProducts).map{caseRating(user,product,rate)=>
((user,product),rate)
}
valratesAndPreds=ratings.map{caseRating(user,product,rate)=>
((user,product),rate)
}.join(predictions)
valMSE=ratesAndPreds.map{case((user,product),(r1,r2))=>
valerr=(r1-r2)
err*err
}.mean()
println("Mean Squared Error = "+MSE)

// Save and load model
model.save(sc,"target/tmp/myCollaborativeFilter")
valsameModel=MatrixFactorizationModel.load(sc,"target/tmp/myCollaborativeFilte
r")
```

完整的实例代码：

```
“examples/src/main/scala/org/apache/spark/examples/mllib/RecommendationExample
.scala”。
```

如果评分矩阵是从其他信息源获得的，可以采用 trainImplicit 算法获得更好的结果。

```
valalpha=0.01
vallambda=0.01
valmodel=ALS.trainImplicit(ratings,rank,numIterations,lambda,alpha)
```

2. Java

ALS Java docs 更多的细节参考 API。

```
importscala.Tuple2;

importorg.apache.spark.api.java.*;
importorg.apache.spark.api.java.function.Function;
importorg.apache.spark.mllib.recommendation.ALS;
importorg.apache.spark.mllib.recommendation.MatrixFactorizationModel;
importorg.apache.spark.mllib.recommendation.Rating;
importorg.apache.spark.SparkConf;

SparkConfconf=newSparkConf().setAppName("Java Collaborative Filtering
Example");
JavaSparkContextjsc=newJavaSparkContext(conf);

// Load and parse the data
Stringpath="data/mllib/als/test.data";
JavaRDD<String>data=jsc.textFile(path);
JavaRDD<Rating>ratings=data.map(
  newFunction<String,Rating>(){
    publicRatingcall(Strings){
      String[]sarray=s.split(",");
      returnnewRating(Integer.parseInt(sarray[0]),Integer.parseInt(sarray[1]),
        Double.parseDouble(sarray[2]));
    }
  }
);

// Build the recommendation model using ALS
```

```
intrank=10;
intnumIterations=10;
MatrixFactorizationModelmodel=ALS.train(JavaRDD.toRDD(ratings),rank,numIterati
ons,0.01);

// Evaluate the model on rating data
JavaRDD<Tuple2<Object,Object>>userProducts=ratings.map(
  newFunction<Rating,Tuple2<Object,Object>>(){
    publicTuple2<Object,Object>call(Ratingr){
      returnnewTuple2<Object,Object>(r.user(),r.product());
    }
  }
);
JavaPairRDD<Tuple2<Integer,Integer>,Double>predictions=JavaPairRDD.fromJavaRDD
(
  model.predict(JavaRDD.toRDD(userProducts)).toJavaRDD().map(
    newFunction<Rating,Tuple2<Tuple2<Integer,Integer>,Double>>(){
      publicTuple2<Tuple2<Integer,Integer>,Double>call(Ratingr){
        returnnewTuple2<>(newTuple2<>(r.user(),r.product()),r.rating());
      }
    }
));
JavaRDD<Tuple2<Double,Double>>ratesAndPreds=
JavaPairRDD.fromJavaRDD(ratings.map(
  newFunction<Rating,Tuple2<Tuple2<Integer,Integer>,Double>>(){
    publicTuple2<Tuple2<Integer,Integer>,Double>call(Ratingr){
      returnnewTuple2<>(newTuple2<>(r.user(),r.product()),r.rating());
    }
  }
)).join(predictions).values();
doubleMSE=JavaDoubleRDD.fromRDD(ratesAndPreds.map(
  newFunction<Tuple2<Double,Double>,Object>(){
    publicObjectcall(Tuple2<Double,Double>pair){
      Doubleerr=pair._1()-pair._2();
      returnerr*err;
    }
  }
).rdd()).mean();
System.out.println("Mean Squared Error = "+MSE);
```

```
// Save and load model
model.save(jsc.sc(),"target/tmp/myCollaborativeFilter");
MatrixFactorizationModelsameModel=MatrixFactorizationModel.load(jsc.sc(),
"target/tmp/myCollaborativeFilter");
```

完整的实例代码：

```
“examples/src/main/java/org/apache/spark/examples/mllib/JavaRecommendationExam
ple.java”。
```

3. Python

ALS Python docs 更多的细节参考 API。

```
frompyspark.mllib.recommendationimportALS,MatrixFactorizationModel,Rating

# Load and parse the data
data=sc.textFile("data/mllib/als/test.data")
ratings=data.map(lambdal:l.split(','))\
.map(lambdal:Rating(int(l[0]),int(l[1]),float(l[2])))

# Build the recommendation model using Alternating Least Squares
rank=10
numIterations=10
model=ALS.train(ratings,rank,numIterations)

# Evaluate the model on training data
testdata=ratings.map(lambdap:(p[0],p[1]))
predictions=model.predictAll(testdata).map(lambdar:((r[0],r[1]),r[2]))
ratesAndPreds=ratings.map(lambdar:((r[0],r[1]),r[2])).join(predictions)
MSE=ratesAndPreds.map(lambdar:(r[1][0]-r[1][1])**2).mean()
print("Mean Squared Error = "+str(MSE))

# Save and load model
model.save(sc,"target/tmp/myCollaborativeFilter")
sameModel=MatrixFactorizationModel.load(sc,"target/tmp/myCollaborativeFilter")
```

完整的实例代码：

```
examples/src/main/python/mllib/recommendation_example.py
```

如果评分矩阵是从其他信息源获得的，可以采用 trainImplicit 算法获得更好的结果：

```
# Build the recommendation model using Alternating Least Squares based on
implicit ratings
model=ALS.trainImplicit(ratings,rank,numIterations,alpha=0.01)
```

5.9.5 Spark MLlib 电影评级推荐

1. 推荐系统常用数据集

（1）MovieLens

MovieLens 数据集中，用户对自己看过的电影进行评分，分值为 1～5。MovieLens 包括两个不同大小的库，适用于不同规模的算法。小规模的库是 943 个独立用户对 1682 部电影做的 10000 次评分的数据；大规模的库是 6040 个独立用户对 3900 部电影做的大约 100 万次评分。

（2）EachMovie

HP/Compaq 的 DEC 研究中心曾经在网上架设 EachMovie 电影推荐系统对公众开放。之后，这个推荐系统关闭了一段时间，其数据作为研究用途对外公布，MovieLens 的部分数据就是来自于这个数据集的。这个数据集有 72916 个用户对 1628 部电影进行的 2811983 次评分。早期大量的协同过滤的研究工作都是基于这个数据集的。2004 年 HP 重新开放 EachMovie，这个数据集就不提供公开下载了。

（3）BookCrossing

这个数据集是网上的 Book-Crossing 图书社区的 278858 个用户对 271379 本书进行的评分，包括显式和隐式的评分。这些用户的年龄等人口统计学属性（demographic feature）都以匿名的形式保存并供分析。这个数据集是由 Cai-Nicolas Ziegler 使用爬虫程序在 2004 年从 Book-Crossing 图书社区上采集的。

（4）Jester Joke

Jester Joke 是一个网上推荐和分享笑话的网站。这个数据集有 73496 个用户对 100 个笑话做的 410 万次评分。评分范围是-10~10 的连续实数。这些数据是由加州大学伯克利分校的 Ken Goldberg 公布的。

（5）Netflix

这个数据集来自于电影租赁网址 Netflix 的数据库。Netflix 于 2005 年底公布此数据集并设立百万美元的奖金（netflix prize），征集能够使其推荐系统性能上升 10%的推荐算法和架构。这个数据集包含了 480189 个匿名用户对大约 17770 部电影作的大约 10 亿次评分。

（6）Usenet Newsgroups

这个数据集包括 20 个新闻组的用户浏览数据。最新的应用是在 KDD2007 上的论文。新闻组的内容和讨论的话题包括计算机技术、摩托车、篮球、政治等。用户们对这些话题进行评价和反馈。

（7）UCI 知识库

UCI 知识库是 Blake 等人在 1998 年开放的一个用于机器学习和评测的数据库，其中存储

大量用于模型训练的标注样本。

（8）http://snap.stanford.edu/na09/resources.html

（9）http://archive.ics.uci.edu/ml/

（10）http://www.ituring.com.cn/article/details/1188

我们将使用 MovieLens 数据集（下载地址：http://www.grouplens.org/node/12）的两个文件：ratings.dat、 movies.dat。格式为：用户 ID，电影 ID，评分，评价时间。所有的评级都包含在 ratings.dat 文件中，并以下格式描述：

```
用户名：：：：：：时间戳 movieid 评级
```

电影信息在文件“movies.dat”中并以下面的格式存在：

```
标题：：：：movieid 流派
```

2. Spark MLlib 电影评级推荐模型（ModelCF）

基于模型的协同过滤推荐就是基于样本的用户喜好信息，训练一个推荐模型，然后根据实时的用户喜好的信息进行预测，计算推荐，如图 5-12 所示。

Low-Rank Matrix Factorization

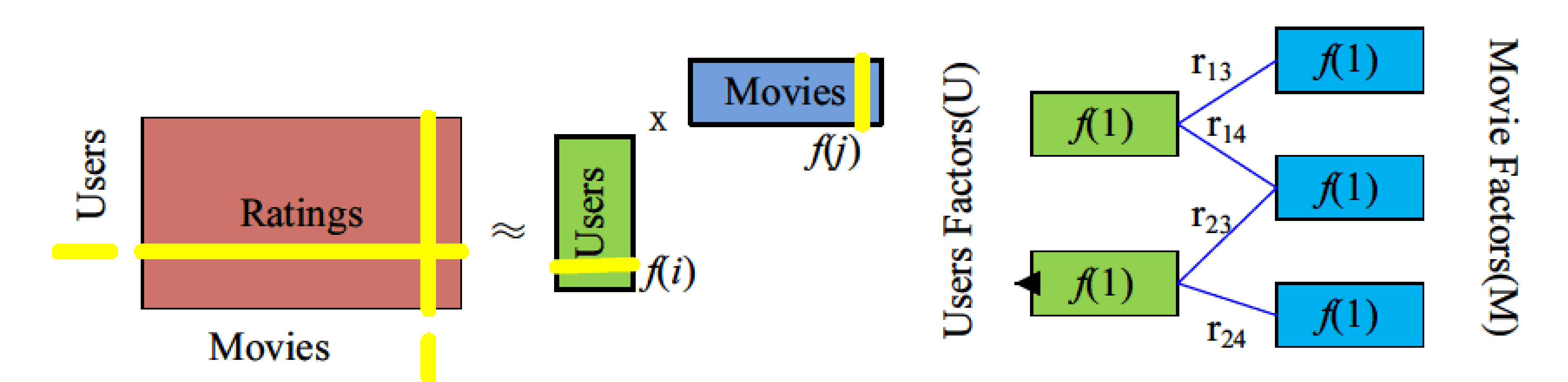


图 5-12　ModelCF

Iterate：

$$f[i] = \arg\min_{\omega \in \mathbb{R}^d} \sum_{j \in Nbrs(i)} (r_{ij} - \omega^T f[j])^2 + \lambda \|\omega\|_2^2 \tag{5-34}$$

3. 创建训练实例

通过运行 bin/rateMovies，在 MovieLens 数据集中选择一小部分用户评级的电影数据形成你的用户评级推荐：

```
python bin/rateMovies
```

当运行这个脚本时，注意看类似下面的提示：

```
Please rate the following movie (1-5 (best), or 0 if not seen):
Toy Story (1995):
```

用 MovieLens 格式，在 personalRatings.txt 中存储用户的电影评级信息，在文件中分配给用户 ID，如果想知道自己的评级对评级推荐的影响，rateMovies 允许重新评级电影。用户如果没安装 python，可以复制 personalRatings.txt.template 到 personalRatings.txt 中，并用评级代替?s 。

4. 以 Scala 为例安装程序

我们将使用一个独立的项目模板训练。训练的 USB 驱动安装在 machine-learning/scala/. 中，在以下目录中找到相应的项目（ items）：

- build.sbt: SBT 项目文件。
- MovieLensALS.scala: 要编辑、编译和运行的主要 Scala 程序。
- Solution: 包含解决方案代码目录。

以下是要编辑、编译的主要文件，并运行。

MovieLensALS.scala 如下：

```
importjava.io.File

importscala.io.Source

importorg.apache.log4j.Logger
import org.apache.log4j.Level

importorg.apache.spark.SparkConf
importorg.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.rdd._
import org.apache.spark.mllib.recommendation.{ALS, Rating,
MatrixFactorizationModel}

object MovieLensALS {

  def main(args: Array[String]) {

    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

    if (args.length != 2) {
      println("Usage: [usb root directory]/spark/bin/spark-submit --driver-
memory 2g --class MovieLensALS " +
```

```
"target/scala-*/movielens-als-ssembly-*.jar movieLensHomeDir
personalRatingsFile")
      sys.exit(1)
    }

    // set up environment

    val conf = new SparkConf()
      .setAppName("MovieLensALS")
      .set("spark.executor.memory", "2g")
    val sc = new SparkContext(conf)

    // load personal ratings

    val myRatings = loadRatings(args(1))
    val myRatingsRDD = sc.parallelize(myRatings, 1)

    // load ratings and movie titles

    val movieLensHomeDir = args(0)

    val ratings = sc.textFile(new File(movieLensHomeDir,
"ratings.dat").toString).map { line =>
      val fields = line.split("::")
      // format: (timestamp % 10, Rating(userId, movieId, rating))
      (fields(3).toLong % 10, Rating(fields(0).toInt, fields(1).toInt,
fields(2).toDouble))
    }

    val movies = sc.textFile(new File(movieLensHomeDir,
"movies.dat").toString).map { line =>
      val fields = line.split("::")
      // format: (movieId, movieName)
      (fields(0).toInt, fields(1))
    }.collect().toMap

    // your code here

    // clean up
```

```
    sc.stop()
  }

  /** Compute RMSE (Root Mean Squared Error). */
  def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating], n: Long):
Double = {
    // ...
  }

  /** Load ratings from file. */
  def loadRatings(path: String): Seq[Rating] = {
    // ...
  }
}
```

先仔细看看文本编辑器中模板代码，然后对模板开始添加代码。定位 movielensals 类，使用文本编辑器打开类。

```
usb/$ cd machine-learning/scala
vim MovieLensALS.scala  # Or your editor of choice
```

任何 Spark 的计算，首先创建一个 sparkconf 对象并使用它来创建一个对象 sparkcontext。由于将使用 Spark 提交程序执行，所以只需要配置执行内存分配并给出该程序的名称，例如“movielensals”，以标识 Spark 的 Web UI。本地模式中，在程序的执行时 Web UI 可以访问本地：4040 端口。这就使它看起来像模板代码：

```
val conf = new SparkConf()
     .setAppName("MovieLensALS")
     .set("spark.executor.memory", "2g")
val sc = new SparkContext(conf)
```

下一步，该代码使用 sparkcontext 读评级。注意，评级文件是以“：：”作为定界符的文本文件。分析每一行代码创建一个评级 RDD，包含（Int, Rating）对，只保留时间戳的最后一位数字作为随机密钥。这个评级 Rating 类封装在元组 (user: Int, product: Int, rating: Double)中。

```
val movieLensHomeDir = args(0)

val ratings = sc.textFile(new File(movieLensHomeDir,
"ratings.dat").toString).map { line =>
     val fields = line.split("::")
     // format: (timestamp % 10, Rating(userId, movieId, rating))
```

```
     (fields(3).toLong % 10, Rating(fields(0).toInt, fields(1).toInt,
fields(2).toDouble))
    }
```

其次，阅读代码中的电影 ID 和标题，把它们收集到的电影 ID 和标题图中。

```
val movies = sc.textFile(new File(movieLensHomeDir,
"movies.dat").toString).map { line =>
val fields = line.split("::")
     // format: (movieId, movieName)
     (fields(0).toInt, fields(1))
    }.collect.toMap
```

现在，先编辑添加代码来获得评级的总结。

```
val numRatings = ratings.count
    val numUsers = ratings.map(_._2.user).distinct.count
val numMovies = ratings.map(_._2.product).distinct.count

    println("Got " + numRatings + "ratings from"
     + numUsers + " users on" + numMovies + " movies.")
```

5. 运行程序

spark-submit 是 Spark 在集群和局部以独立模式运行应用推荐的方式。

```
usb/$ cd machine-learning/scala

# The following command compiles the MovieLensALS class
# and creates a jar file in machine-learning/scala/target/scala-2.10/
[usb root directory]/sbt/sbt assembly

# change the folder name from "medium" to "large" to run on the large data set
[usb root directory]/spark/bin/spark-submit --class MovieLensALS target/scala-
2.10/movielens-als-
assembly-0.1.jar [usb root
directory]/data/movielens/medium/ ../personalRatings.txt
```

在屏幕上可以看到类似的输出：

```
Got 1000209 ratings from 6040 users on 3706 movies.
```

利用 MLlib's ALS 训练 matrixfactorizationmodel，需要把 Scala 的 RDD[Rating]和 Python 的 RDD[(user, product, rating)]对象作为输入。ALS 训练参数如矩阵排列因素和常量，用以确

定一个优良组合训练参数，把数据分为三个非重叠的子集，命名为训练、测试和验证，附加时间戳最后数字。根据训练集训练多个模型，对验证集基于均方根误差 RMSE（Root Mean Squared Error）选择最佳模型， 最后在测试集上评价最好的训练模型。还可以添加用户评级到训练集建议用户训练。为多次访问需要在内存中调用 cache 坚持训练和验证测试集。

```
val numPartitions = 4
val training = ratings.filter(x => x._1 < 6)
     .values
     .union(myRatingsRDD)
     .repartition(numPartitions)
     .cache()
val validation = ratings.filter(x => x._1 >= 6 && x._1 < 8)
     .values
     .repartition(numPartitions)
     .cache()
val test = ratings.filter(x => x._1 >= 8).values.cache()

val numTraining = training.count()
val numValidation = validation.count()
val numTest = test.count()

   println("Training: " + numTraining + ", validation: " + numValidation + ",
test: " + numTest)
```

拆分之后，应该看到：

```
Training: 602251, validation: 198919, test: 199049.
```

6. 使用 ALS 训练

在本例中，我们将使用 als.train 训练一堆模型，并从中选择评价最好的。ALS 的训练参数中，最重要的是排名（ rank）、λ（lambda 规范化常量）和迭代次数（iterations）。ALS 的 train 方法我们打算使用的定义如下：

```
object ALS {

def train(ratings: RDD[Rating], rank: Int, iterations: Int, lambda: Double)
   : MatrixFactorizationModel = {
   // ...
  }
}
```

理想的情况是，我们尝试从一大批它们的组合中找到最好的。由于时间的关系，我们将只测试从 2 个不同排列的向量积（8 和 12）构造的 8 个组合，2 个不同 λ（1 和 10）和两种不同的数字（10 和 20）的迭代。对于每种模型，我们用 Spark 提供的方法 computermse，在验证集上计算均方根误差（RMSE）。计算每个模型验证集的均方根误差，选择测试集的均方根误差（RMSE）作为最终的度量。解决方案代码如下：

```
val ranks = List(8, 12)
 val lambdas = List(1.0, 10.0)
 val numIters = List(10, 20)
var bestModel: Option[MatrixFactorizationModel] = None
 var bestValidationRmse = Double.MaxValue
var bestRank = 0
 var bestLambda = -1.0
var bestNumIter = -1
   for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
val model = ALS.train(training, rank, numIter, lambda)
val validationRmse = computeRmse(model, validation, numValidation)
     println("RMSE (validation) = " + validationRmse + " for the model trained
with rank = "
       + rank + ", lambda = " + lambda + ", and numIter = " + numIter + ".")
     if (validationRmse < bestValidationRmse) {
       bestModel = Some(model)
       bestValidationRmse = validationRmse
       bestRank = rank
       bestLambda = lambda
       bestNumIter = numIter
     }
   }

val testRmse = computeRmse(bestModel.get, test, numTest)

   println("The best model was trained with rank = " + bestRank + " and lambda
= "+ bestLambda
+", and numIter = "+ bestNumIter +", and its RMSE on the test set is"+
testRmse +".")
```

Spark 可能会花一两分钟来训练模型，这时你应该看到屏幕上下面的内容：

```
The best model was trained using rank 8 and lambda 10.0, and its RMSE on test
is 0.8808492431998702.
```

7. 电影评级推荐

接下来让我们看一看评级模型为你推荐什么样的电影。对所有你没有评分的电影通过生成（0，movieid）参数对，调用模型的预测 （predict）算法预测的结果。0 是分配给你的特殊用户 ID。

```
class MatrixFactorizationModel {
def predict(userProducts: RDD[(Int, Int)]): RDD[Rating] = {
    // ...
  }
}
```

计算得到所有预测结果后，列出排名前 50 的推荐，看是否符合你的偏好。

```
val myRatedMovieIds = myRatings.map(_.product).toSet
val candidates =
sc.parallelize(movies.keys.filter(!myRatedMovieIds.contains(_)).toSeq)
 val recommendations = bestModel.get
      .predict(candidates.map((0, _)))
      .collect()
      .sortBy(- _.rating)
      .take(50)

var i = 1
    println("Movies recommended for you:")
    recommendations.foreach { r =>
      println("%2d".format(i) + ": " + movies(r.product))
      i += 1
    }
```

相似性结果输出：

```
Movies recommended for you:
 1: Silence of the Lambs, The (1991)
 2: Saving Private Ryan (1998)
 3: Godfather, The (1972)
 4: Star Wars: Episode IV - A New Hope (1977)
 5: Braveheart (1995)
 6: Schindler's List (1993)
 7: Shawshank Redemption, The (1994)
 8: Star Wars: Episode V - The Empire Strikes Back (1980)
 9: Pulp Fiction (1994)
```

```
10: Alien (1979)
...
```

以上是利用老数据集做的推荐，所以电影年份上看有些老。

ALS 输出一个非平凡的影评模型吗？评级结果可以与平凡基准模型输出的平均评分比较（或者你可以尝试输出的每部电影的平均等级）。计算基准的均方根误差（RMSE）直截了当。解决方案如下代码：

```
val meanRating = training.union(validation).map(_.rating).mean
val baselineRmse =
     math.sqrt(test.map(x => (meanRating - x.rating) * (meanRating -
x.rating)).mean)
val improvement = (baselineRmse - testRmse) / baselineRmse * 100
   println("The best model improves the baseline by " +
"%1.2f".format(improvement) + "%.")
```

类似输出：

```
The best model improves the baseline by 20.96%.
```

这似乎是显而易见的，训练好的模型会胜过平凡基准。然而，一个糟糕的训练参数组合会导致模型比平凡基准差。选择正确的训练参数集是任务的关键。

一个更好的方法是首先为你的推荐训练集构建分解矩阵模型，添加你的用户评级到训练集，然后用你的评级增强模型，你可以执行 MatrixFactorizationModel 看是否模型对于新用户有新的更新。

5.10 Spark MLlib 神经网络算法

神经网络在一定程度上受到生物学的启发，由一系列相互链接的神经单元组成，每一个单元都有一定数量的实值输入（可能由其他神经单元输出），并产生单一的实数值输出（可能成为其他很多单元的输入）[90,91]，如图 5-13 所示（来源网络）。

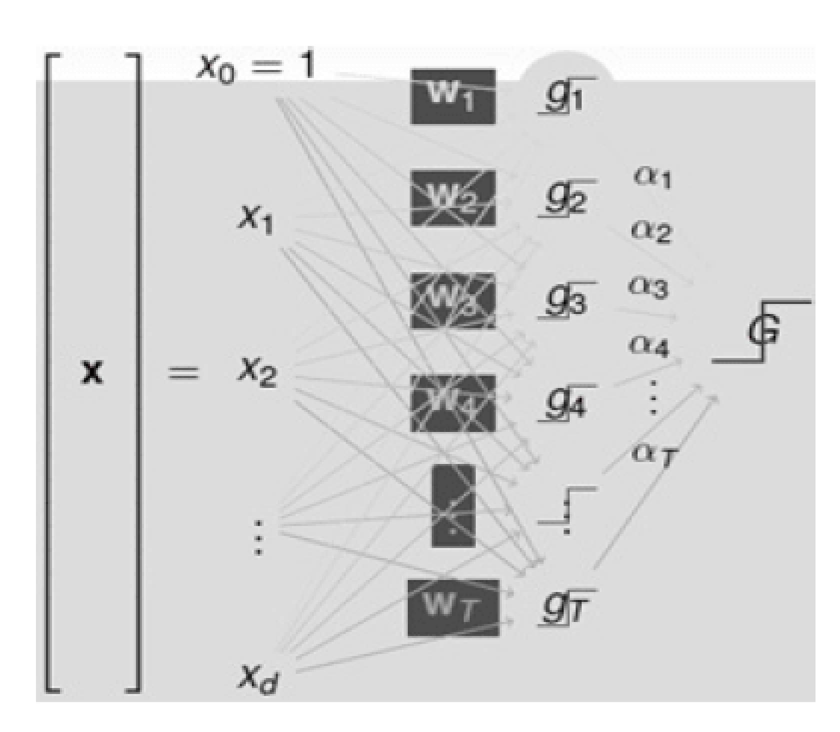


图 5-13　神经网络系统原理

其中，$g_1,g2,...,g_T$ 的输入为：$X_0,X_1,...,X_d$，而 $g_1,g_2,...,g_T$ 又作为 G 的输入。

神经网络系统的一个基本单元被称为一个感知机，如图 5-14 所示，以一个实数值向量作为输入，计算这些输入的线性组合，如果结果大于 0，就输出 1，否则输出-1，如表达式 5-35 所示[92,93]：

$$h(x) = sign(\sum_{n=0}^{d} w_n x_n) = sign(w^T x) \qquad (5\text{-}35)$$

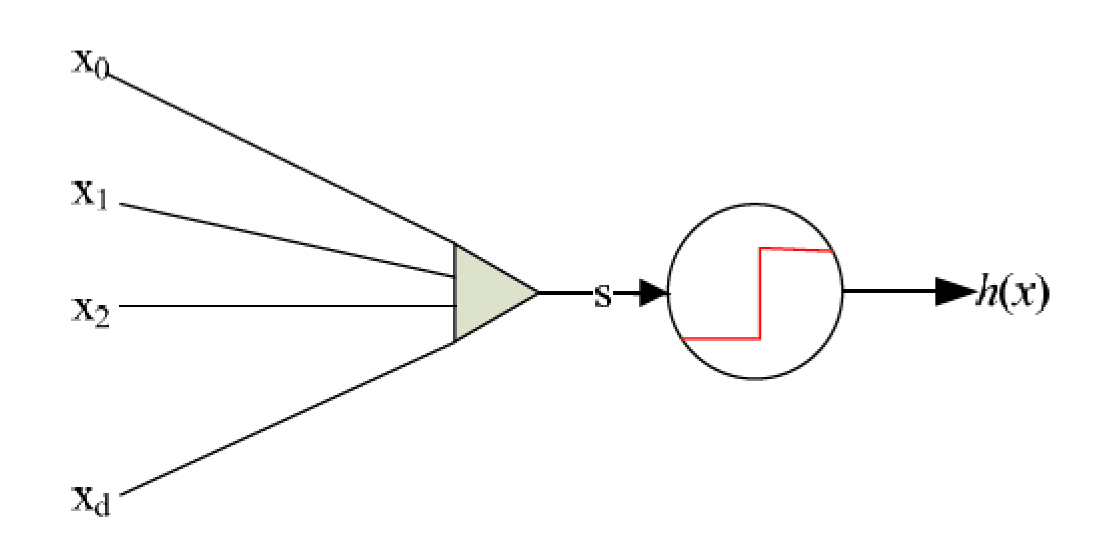


图 5-14　感知机

单独的一个感知机可以用来表示原子布尔函数，作为对一个超平面进行划分的线性分割面。然而，很多样例不是线性可分的，可能需要多个分割面组成，如图 5-15（来源网络）所示，圆中的点与圆外的点，只用一个线性分割面是无法分开的，如果样本数继续增加，那么就需要更多的感知器来逼近实际的分割面[94]。

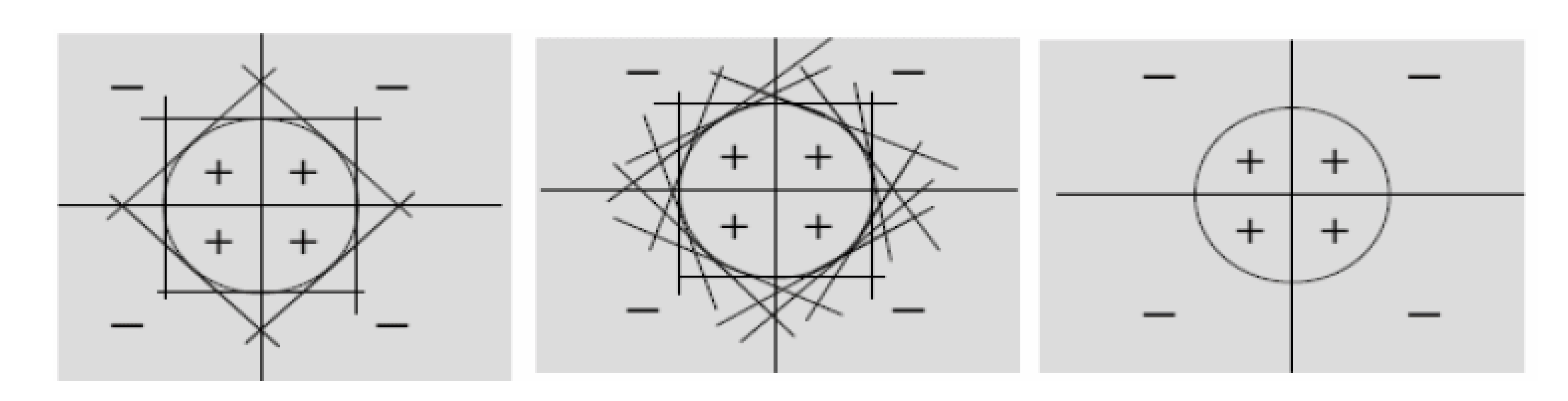


图 5-15 多样本感知机逼近线性分割面（8 个感知机、16 个感知机和目标边界）

由神经网络系统原理得：

$$g_1 = sign(w_1^T x)$$

$$g_2 = sign(w_2^T x)$$

$$\cdots$$

$$g_T = sign(w_T^T x) \qquad (5\text{-}36)$$

然后得：

$$G = sign(\sum_{t=1}^{T} \alpha_t g_t) = sign(\alpha^T g) \qquad (5\text{-}37)$$

最后得：

$$G = sign(\sum_{t=1}^{T} \alpha_t sign(w_t^T x)) \tag{5-38}$$

针对不同的问题感知机可能有不同的形式，如图 5-16 所示，分别为：线性分类、线性回归、逻辑回归。

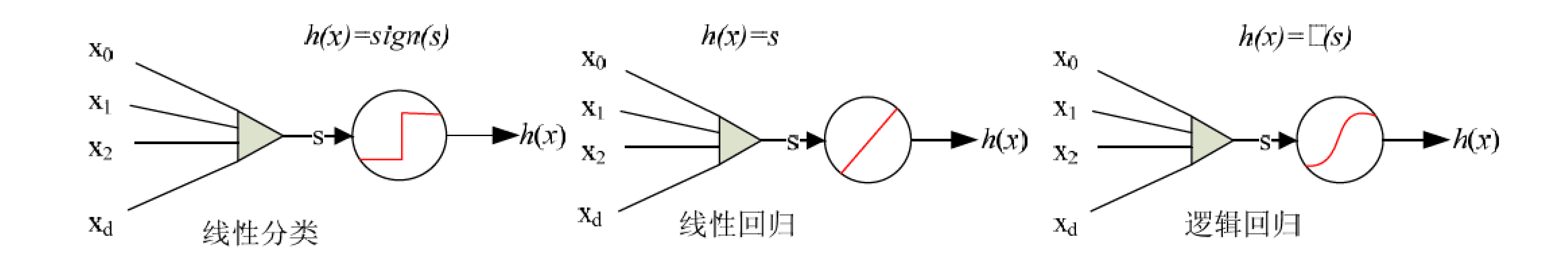


图 5-16　变换函数

针对线性分类，$h(x)$ 不可微，很难通过优化求解方式来得到 w。

针对线性回归，$h(x)=s$ 可以表示为 5-39 式的形式，整个网络都是线性的，也就失去了多层网络的意义。

$$G = [\alpha_1 \alpha_2 \dots \alpha_T] \left(\begin{bmatrix} w_{10} & w & \dots & w_{1d} \\ w_{20} & w & \dots & \dots \\ \dots & \dots & \dots & \dots \\ w_{T0} & \dots & \dots & w_{Td} \end{bmatrix} * \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_d \end{bmatrix} \right) = \alpha(wx) = \beta x \tag{5-39}$$

针对逻辑回归，常用的为 sigmoid 函数，也可以称为 logistic 函数，$\sigma(y)=1/(1+e^{-y})$，它的范围为 0 到 1，通常也会将其平移缩放到 -1 到 1 的范围：

$$\delta(s) = 2\delta(s) - 1 = \frac{e^s - e^{-s}}{e^s + e^{-s}} = \tanh(s) \tag{5-40}$$

一个典型的神经网络，假设总共有 L 层（从 0 层开始）。其中的输入数据 x 称为输入层 $x^{(0)}$，表示第 $d^{(0)}$ 层；最后一层的 $x^{(L)}{}_1$ 称为输出层，表示第 $d^{(L)}$ 层；中间的这些层表示隐藏层。

权重为：

$$w_{ij}^l : \begin{cases} 1 \le l \le L & layers \\ 0 \le i \le d^{(l-1)} & inputs \\ 1 \le j \le d^l & outputs \end{cases} \tag{5-41}$$

分数为：

$$s_j^l = \sum_{i=0}^{d^{(l-1)}} w_{ij}^l x_i^{(l-1)} \tag{5-42}$$

变换函数为：

$$_j^{(l)} = \begin{cases} \tanh(s_j^{(l)}) & l \le L \\ s_j^{(l)} & l = L \end{cases} \tag{5-43}$$

每一层的变换都可以是先进行权重矩阵与输入向量的乘法，然后得到新的向量，再对新的向量中的每一个数据进行 tanh 处理。

$$\begin{aligned} \phi^{(l)}(x) &= \tanh\left(\begin{bmatrix} s_1^{(l)} \\ \cdots \\ s_{d^{(l)}}^{(l)} \end{bmatrix}\right) \\ &= \tanh\left(\begin{bmatrix} \sum_{i=0}^{d^{(l-1)}} w_{i1}^{(l)} x_i^{(l-1)} \\ \cdots \\ \sum_{i=0}^{d^{(l-1)}} w_{id^{(l)}}^{(l)} x_i^{(l-1)} \end{bmatrix}\right) \\ &= \tanh\left(\begin{bmatrix} w_{01}^{(l)} & \cdots & w_{d^{(l-1)}1}^{(l)} \\ \cdots & \cdots & \cdots \\ w_{0d^{(l)}}^{(l)} & \cdots & w_{d^{(l-1)}d^{(l)}}^{(l)} \end{bmatrix} * \begin{bmatrix} x_0^{(l-1)} \\ \cdots \\ x_{d^{(l-1)}}^{(l-1)} \end{bmatrix}\right) \\ &= \tanh(w^{(l)} x^{(l-1)}) \end{aligned} \tag{5-44}$$

最后得：

$$y = w^{(l)}(\tanh(w^{(L-1)}(\ldots \tanh(w^{(1)} x^{(0)})))) \tag{5-45}$$

Spark MLlib NeuralNet 代码参见附录，可从下面网址下载。
https://github.com/sunbow1/SparkMLlibDeepLearn

5.11 本章小结

本章主要从 Spark MLlib 的大数据算法实现的角度，集中概括了 Spark 机器学习的分类及其案例分析。然后针对常用的 Spark MLlib 机器学习的算法分别以 Scala、R 和 Java 语言给出了算法实现。

第 6 章

Spark大数据架构系统部署

数据管理比以往更加复杂，到处都是大数据，包括每个人的想法以及不同的形式：广告、社交图谱、信息流、推荐、市场、健康、 安全、政府等。过去的几年里，成千上万的技术必须处理汇合在一起的大数据获取、管理和分析等业务。技术选型对 IT 部门来说是一件艰巨的任务，因为在大多数时间里没有一个综合的方法来用于选型。

大数据我们也许听得很多了，一般都知道 Hadoop，但并不都是 Hadoop。那么我们该如何构建自己的大数据项目呢？对于离线处理，Hadoop 还是比较适合的，但对于实时性比较强的，数据量比较大的，可以采用 Spark，那 Spark 又跟什么技术搭配才能做一个适合自己的项目呢？各技术之间又是如何整合的呢？本章目的是定义大数据的表征，将给大家介绍下大数据项目中用到的各种技术框架知识，并通过一个实际项目的分布式集群部署和实际业务应用来详细讲述大数据架构是如何构建的。

6.1 大数据架构介绍

大数据可通过许多方式来存储、获取、处理和分析。每个大数据来源都有不同的特征，包括数据的频率、量、速度、类型和真实性。处理并存储大数据时，会涉及更多维度，比如治理、安全性和策略。选择一种架构并构建合适的大数据解决方案极具挑战性，因为需要考虑非常多的因素[95]。大数据处理流程如图 6-1 所示。

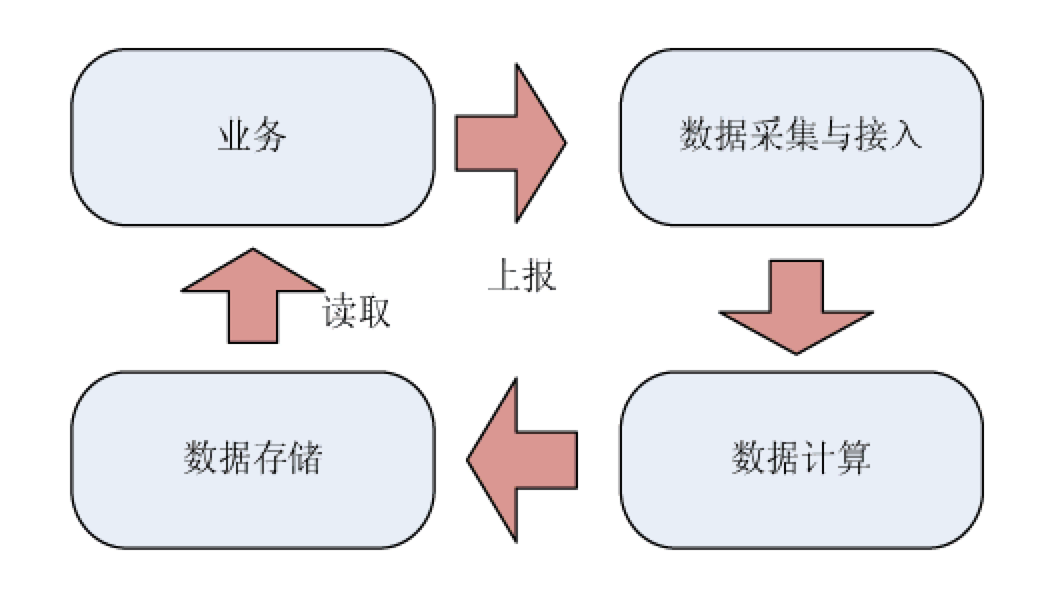


图 6-1 大数据处理流程

业务问题可分类为不同的大数据问题类型。将使用此类型确定合适的分类模式和合适的大数据解决方案。第一步是将业务问题映射到它的大数据类型。

按类型对大数据问题进行分类，更容易看到每种数据的特征。这些特征可帮助我们了解

如何获取数据、如何将它处理为合适的格式，以及新数据出现的频率。来自不同来源的数据具有不同的特征。例如，社交媒体数据包含不断传入的视频、图像和非结构化文本，对数据进行分类后，就可以将它与合适的大数据模式匹配[96]。

需要一个架构具备长时间处理和准实时数据处理的能力。这一架构是分布式的，而不是依赖于高性能且价格高昂的商用机，取而代之的是高可用、性能驱动和廉价技术所赋予的灵活性[97]。

6.2 典型的商务使用场景

除了技术和架构考虑，需要面对典型大数据用例的使用场景。它们部分与特殊的工业领域相关，另外的部分可能适应于各种领域。这些考虑一般都是基于分析应用的日志，例如 Web 访问日志、应用服务器日志和数据库日志，但是也可以基于各种其他的数据源，例如社交网络数据。当面对这些使用场景的时候，如果希望随着商务的增长而弹性扩展，就需要考虑设计一个分布式的大数据架构。

6.2.1 客户行为分析

感知客户，或者叫做 “360 度客户视角”可能是最流行的大数据使用场景。客户视角通常用于电子商务网站，一般它始于一个非结构化的点击流，换而言之，由一个访客执行的主动点击和被动的网站导航操作组成。通过计算和分析点击量和面向产品或广告的印象，可以依赖行为而适配访客的用户体验，目标是得到优化漏斗转换的模型。

采用标签库体系理论，结合大数据分析和实践方法，基于来自于企业内外部的客户、产品、销售、资产、行为轨迹、体验评价属性等相关数据，通过数据挖掘/机器学习等，提取出有价值的信息，形成客户统一视图，构建符合业务规则的客户标签体系。

客户标签体系从客户基本信息、客户行为轨迹等多个维度，为每一个客户打上特定的标签，通过标签可识别特定特征的目标群体，使得定位目标群体更加准确、高效。一方面高效快速地增加企业收入的同时节约企业的运营成本、降低交易风险，另一方面增强客户体验，提升客户满意度、客户忠诚度。客户行为分析图如图 6-2 所示。

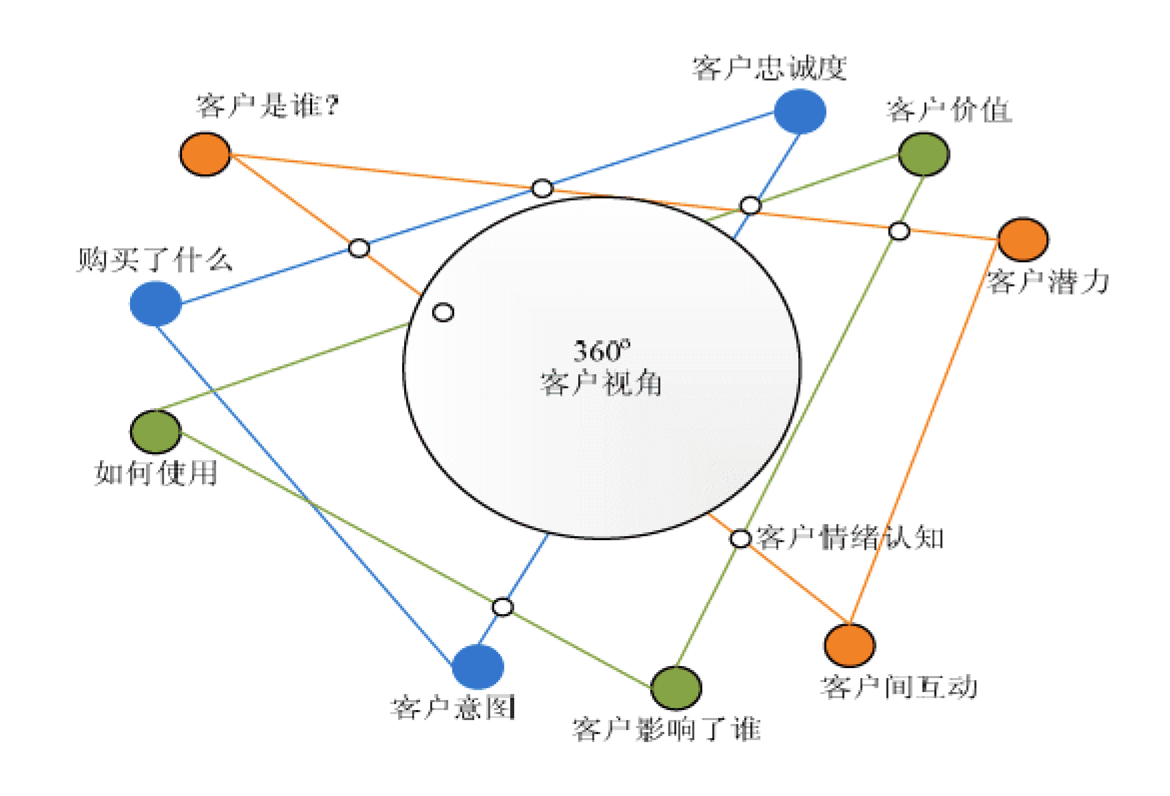


图 6-2　客户行为分析图

1. 数据采集

建立客户信息采集渠道网络，从业务系统、数据库、互联网、社交媒体等各种数据源，采集用户数据，包括但不限于人口统计学特征数据、兴趣特征数据、社会属性特征数据、业务特征数据等。

2. 挖掘分析

整合数据，通过数据清洗、预处理等基本步骤，利用业务规则归纳、统计分析、数学建模、机器学习等手段挖掘提取有价值的信息。

3. 标签体系建设

基于挖掘得出的有价值信息，结合企业业务规则，构建完善的客户标签体系。

4. 标签应用

将已经开发好的标签应用到实际的营销、管理工作中。

5. 效果监控

提供可视化的界面，监控已经上线标签的应用效果。

6.2.2　情绪分析

商务应用关注的是其在社交网络上所被感知的形象和声誉，把可能使他们声名狼藉的负面事件最小化并充分利用正面事件。通过准实时抓取大量的社交数据，可以提取出社交社区中关于品牌的感受和情绪，从而影响用户并联系他们，改变并强化与这些用户的交互[98]。

对于面向过程的服务类型、服务质量，却很难以结构化的数据去评价和判断。客户对整个服务过程的主观评价，成为研究服务质量的重要来源。客户对整体服务质量的评价，很大程度上会转化为情绪释放，因此，客户情绪在服务质量评价中具有重要的研究意义[99]。

在多元化营销服务的大环境下，不同客户对服务质量要求的不同，不同的服务质量因素、标准对不同客户的影响也截然不同。在客户反馈的过程中，客户不同的情绪很大程度上反映了对企业的满意度和忠诚度，在这个角度，探索客户情绪与产品服务质量的关系，对防止客户悄然流逝，预先进行质量预控会有更大的帮助和效果。在分析影响客户情绪的因素中，结合客户情绪与服务满意度的关系理论，从服务的内、外部因素着手，包括市场竞争环境、客户生命周期、服务本身的特征等方面对情绪的影响，总结了多种条件下客户表达情绪的规律[100]。

利用数据挖掘方法、文本识别模型智能的对非结构化的客户评价文本进行自动识别，并从中挖掘客户释放的情绪特征，以及影响客户服务质量的要素，通过建立评估规则和预控模型，形成对服务质量风险的评价和措施。

6.2.3　CRM Onboarding

基于访客的社交行为，可以将客户的行为分析和数据的情感分析结合在一起。希望将这

些在线数据源和已经存在的离线数据结合在一起，这叫做 CRM （customer relationship management）onboarding，以便于得到更好和更准确的客户定位。进而，公司能够充分利用这一定位，从而建立更好的目标系统，使市场活动的效益最大化。

6.2.4 预测

从数据中学习在过去几年已经成为主要的大数据趋势。基于大数据的预测在许多业界是非常有效的，例如百度预测、经济指数预测、景点预测、疾病预测、城市预测、欧洲赛事预测、世界杯预测、高考预测、电影票房预测等。预测性分析是大数据最核心的功能。

大数据还拥有数据可视化和大数据挖掘的功能，对已发生的信息价值进行挖掘并辅助决策。传统的数据分析挖掘在做相似的事情，只不过效率会低一些或者说挖掘的深度、广度和精度不够。大数据预测则是基于大数据和预测模型去预测未来某件事情的概率。让分析从“面向已经发生的过去”转向“面向即将发生的未来”，这是大数据与传统数据分析的最大不同。

大数据预测的逻辑基础是，每一种非常规的变化事前一定有征兆，每一件事情都有迹可循，如果找到了征兆与变化之间的规律，就可以进行预测。大数据预测无法确定某件事情必然会发生，它更多的是给出一个概率。

6.3 Spark 三种分布式部署模式

目前 Apache Spark 支持三种分布式部署方式，分别是 standalone、spark on Mesos 和 spark on YARN，其中，第一种类似于 MapReduce 1.0 所采用的模式，内部实现了容错性和资源管理，后两种则是未来发展的趋势，部分容错性和资源管理交由统一的资源管理系统完成：让 Spark 运行在一个通用的资源管理系统之上，这样可以与其他计算框架，比如 MapReduce，共用一个集群资源，最大的好处是降低运维成本和提高资源利用率（资源按需分配）。

6.3.1 Standalone 模式

Standalone 模式，即独立模式，自带完整的服务，可单独部署到一个集群中，无须依赖任何其他资源管理系统。从一定程度上说，该模式是其他两种的基础。借鉴 Spark 开发模式，我们可以得到一种开发新型计算框架的一般思路：先设计出它的 standalone 模式，为了快速开发，起初不需要考虑服务（比如 master/slave）的容错性，之后再开发相应的 wrapper，将 standalone 模式下的服务原封不动地部署到资源管理系统 YARN 或者 Mesos 上，由资源管理系统负责服务本身的容错。目前 Spark 在 standalone 模式下是没有任何单点故障问题的，这是借助 zookeeper 实现的，思想类似于 HBase master 单点故障解决方案。将 Spark standalone 与 MapReduce 比较，会发现它们两个在架构上是完全一致的：

（1）都是由 master/slaves 服务组成的，且起初 master 均存在单点故障，后来均通过

zookeeper 解决（Apache MRv1 的 JobTracker 仍存在单点问题，但 CDH 版本得到了解决）。

（2）各个节点上的资源被抽象成粗粒度的 slot，有多少 slot 就能同时运行多少 task。不同的是，MapReduce 将 slot 分为 map slot 和 reduce slot，它们分别只能供 Map Task 和 Reduce Task 使用，而不能共享，这是 MapReduce 资源利率低效的原因之一，而 Spark 则更优化一些，它不区分 slot 类型，只有一种 slot，可以供各种类型的 Task 使用，这种方式可以提高资源利用率，但是不够灵活，不能为不同类型的 Task 定制 slot 资源。

总之，这两种方式各有优缺点。

6.3.2 Spark On Mesos 模式

Spark On Mesos 模式是很多公司采用的模式，官方推荐这种模式（当然，原因之一是血缘关系）。正是由于 Spark 开发之初就考虑到支持 Mesos，因此，目前而言，Spark 运行在 Mesos 上会比运行在 YARN 上更加灵活，更加自然。目前在 Spark On Mesos 环境中，用户可选择两种调度模式之一运行自己的应用程序（可参考 Andrew Xia 的《Mesos Scheduling Mode on Spark》）。

（1）粗粒度模式（Coarse-grained Mode）。每个应用程序的运行环境由一个 dirver 和若干个 Executor 组成，其中，每个 Executor 占用若干资源，内部可运行多个 Task（对应多少个 slot）。应用程序的各个任务正式运行之前，需要将运行环境中的资源全部申请好，且运行过程中要一直占用这些资源，即使不用，最后程序运行结束后，回收这些资源。举个例子，比如你提交应用程序时，指定使用 5 个 Executor 运行你的应用程序，每个 Executor 占用 5GB 内存和 5 个 CPU，每个 Executor 内部设置了 5 个 slot，则 Mesos 需要先为 Executor 分配资源并启动它们，之后开始调度任务。另外，在程序运行过程中，Mesos 的 master 和 slave 并不知道 Executor 内部各个 task 的运行情况，Executor 直接将任务状态通过内部的通信机制汇报给 driver。从一定程度上可以认为，每个应用程序利用 Mesos 搭建了一个虚拟集群自己使用。

（2）细粒度模式（Fine-grained Mode）。鉴于粗粒度模式会造成大量资源浪费，Spark On Mesos 还提供了另外一种调度模式：细粒度模式，这种模式类似于现在的云计算，思想是按需分配。与粗粒度模式一样，应用程序启动时，先会启动 Executor，但每个 Executor 占用资源仅仅是自己运行所需的资源，不需要考虑将来要运行的任务，之后，Mesos 会为每个 executor 动态分配资源，每分配一些，便可以运行一个新任务，单个 Task 运行完之后可以马上释放对应的资源。每个 Task 会汇报状态给 Mesos slave 和 Mesos master，便于更加细粒度管理和容错，这种调度模式类似于 MapReduce 调度模式，每个 Task 完全独立，优点是便于资源控制和隔离，但缺点也很明显，短作业运行延迟大。

6.3.3 Spark On YARN 模式

Spark On YARN 模式是一种最有前景的部署模式。但限于 YARN 自身的发展，目前仅支持粗粒度模式（Coarse-grained Mode）。这是由于 YARN 上的 Container 资源是不可以动态伸缩的，一旦 Container 启动之后，可使用的资源不能再发生变化，不过这个已经在 YARN 计划（具体参考 https://issues.apache.org/jira/browse/YARN-1197）中了。

当在 YARN 上运行 Spark 作业，每个 Spark executor 作为一个 YARN 容器（container）运行。Spark 可以使得多个 Tasks 在同一个容器里面运行，这是个很大的优点。注意，这里和 Hadoop 的 MapReduce 作业不一样，MapReduce 作业为每个 Task 开启不同的 JVM 来运行，虽然说 MapReduce 可以通过参数来配置。

从广义上讲，yarn-cluster 适用于生产环境，而 yarn-client 适用于交互和调试，也就是希望快速地看到 application 的输出。

在介绍 yarn-cluster 和 yarn-client 的深层次的区别之前，先明白一个概念：Application Master。在 YARN 中，每个 Application 实例都有一个 Application Master 进程，它是 Application 启动的第一个容器。它负责和 ResourceManager 打交道，并请求资源，获取资源之后告诉 NodeManager 为其启动 container。

从深层次的含义讲，yarn-cluster 和 yarn-client 模式的区别其实就是 Application Master 进程的区别。yarn-cluster 模式下，driver 运行在 AM（Application Master）中，它负责向 YARN 申请资源，并监督作业的运行状况。当用户提交了作业之后，就可以关掉 Client，作业会继续在 YARN 上运行。然而 yarn-cluster 模式不适合运行交互类型的作业，而 yarn-client 模式下，Application Master 仅仅向 YARN 请求 executor，client 会和请求的 container 通信来调度他们工作，也就是说 Client 不能离开。

6.4 创建大数据架构

从高层视角来看，我们的架构看起来像另一个电子商务应用架构，需要如下。

- 一个 Web 应用：访客可以用它导航一个产品目录。
- 一个日志摄取应用：提取日志并处理它们。
- 一个机器学习应用：为访客触发推荐。
- 一个处理引擎：作为该架构的中央处理集群。
- 一个搜索引擎：拉取处理数据的分析。

6.4.1 数据采集

数据的获取或者摄取开始于不同的数据源，可能是大的日志文件、流数据、ETL 处理过的输出、在线的非结构化数据，或者离线的结构化数据。负责从各节点上实时采集数据，可以选用 Apache Flume 来实现。

Apache Flume 是 Cloudera（Hadoop 数据管理软件与服务提供商）提供的一个分布式、可靠的、高可用的海量日志采集、聚合和传输的日志收集系统，支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume 提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。Flume 数据流模型如图 6-3 所示。

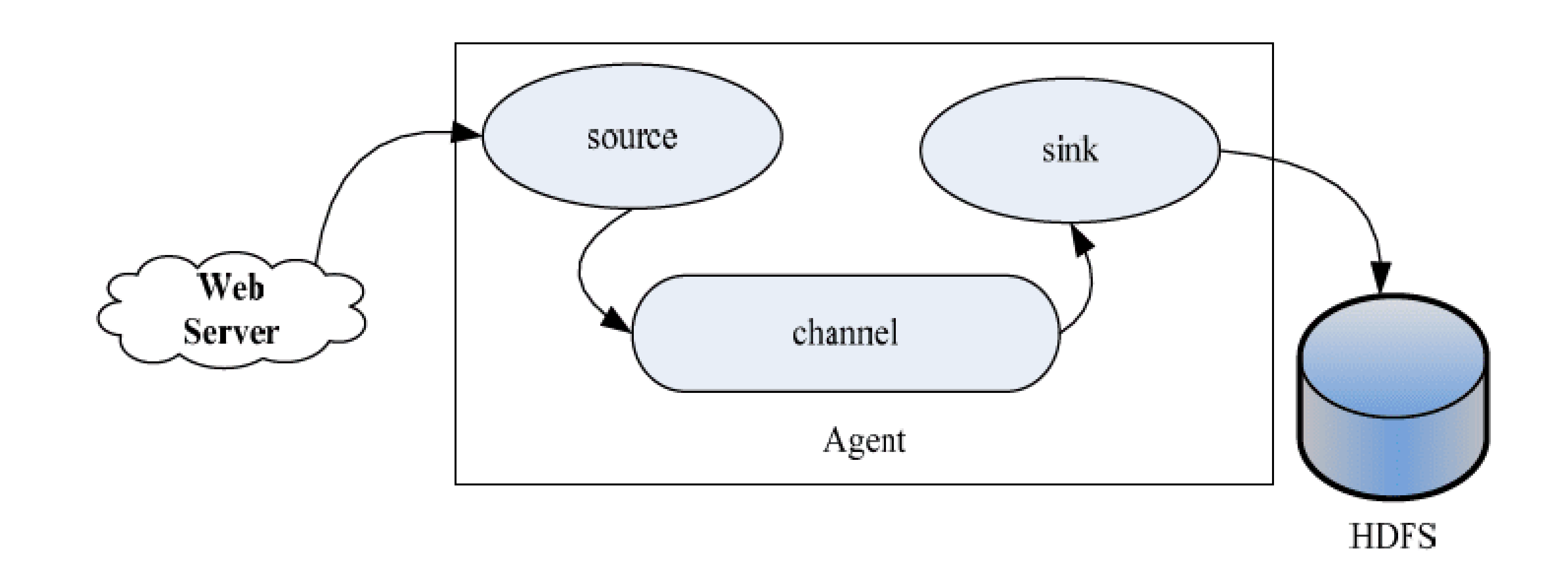


图 6-3　Flume 数据流模型

Flume 的一些核心概念如表 6-1 所示。Flume 以 Agent 为最小的独立运行单位，一个 Agent 就是一个 JVM（JavaVirtual Machine），单 Agent 由 Source、Sink 和 Channel 三大组件构成，Flume 的数据流由事件（Event）贯穿始终。事件是 Flume 的基本数据单位，它携带日志数据（字节数组形式）并且携带有头信息，这些 Event 由 Agent 外部的 Client，比如上图中的 WebServer 生成。当 Source 捕获事件后会进行特定的格式化，然后 Source 会把事件推入（单个或多个）Channel 中。你可以把 Channel 看作是一个缓冲区，它将保存事件直到 Sink 处理完该事件。Sink 负责持久化日志或者把事件推向另一个 Source。

表 6-1　Flume 的一些核心概念

组件	功能
Agent	使用 JVM 运行 Flume。每台机器运行一个 Agent，但是可以在一个 Agent 中包含多个 Sources 和 Sinks
Client	生产数据，运行在一个独立的线程
Source	从 Client 收集数据，传递给 Channel
Sink	从 Channel 收集数据，运行在一个独立线程
Channel	连接 Sources 和 Sinks，这个有点像一个队列
Events	可以是日志记录、avro 对象等

6.4.2　数据接入

由于采集数据的速度和数据处理的速度不一定同步，因此添加一个消息中间件来作为缓冲，选用 Apache 的 Kafka，对于离线数据，选用 HDFS。

1. Kafka

Kafka 是一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者规模的网站中的所有动作流数据。这种动作（网页浏览、搜索和其他用户的行动）是在现代网络上的许多社会功能的一个关键因素。这些数据通常由于吞吐量的要求而需要通过处理日志和日志聚合来解决。对于像 Hadoop 的一样的日志数据和离线分析系统，但又要求实时处理，这是一个可行的解决方案。Kafka 的目的是通过 Hadoop 的并行加载机制来统一线上和离线的消息处理，也是为了通过集群机来提供实时的消费。Kafka 的一些核心概念如表 6-2 所示，Kafka 拓扑结构如图 6-4 所示。

表 6-2　Kafka 的一些核心概念

术语	功能
Broker	Kafka 集群包含一个或多个服务器，这种服务器被称为 broker
Topic	每条发布到 Kafka 集群的消息都有一个类别，这个类别被称为 topic。（物理上不同 topic 的消息分开存储，逻辑上一个 topic 的消息虽然保存在一个或多个 broker 上，但用户只需指定消息的 topic 即可生产或消费数据而不必关心数据存在于何处）
Partition	Partition 是物理上的概念，每个 topic 包含一个或多个 partition
Producer	负责发布消息到 Kafka broker
Consumer	消息消费者，向 Kafka broker 读取消息的客户端
Consumer Group	每个 consumer 属于一个特定的 consumer group （可为每个 consumer 指定 group name，若不指定 group name，则属于默认的 group）

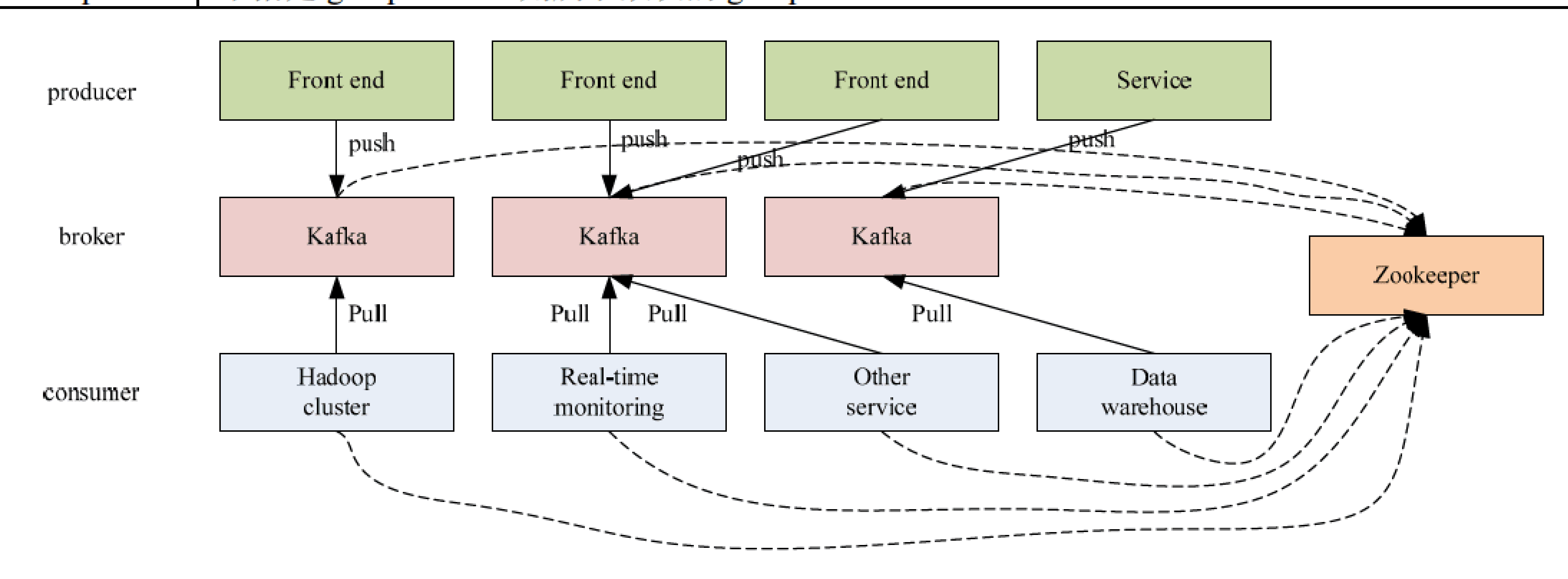


图 6-4　Kafka 拓扑结构

一个典型的 Kafka 集群中包含若干 Producer（可以是 Web 前端产生的 PageView，或者是服务器日志、系统 CPU、Memory 等），若干 broker（Kafka 支持水平扩展，一般 broker 数量越多，集群吞吐率越高），若干 Consumer Group，以及一个 Zookeeper 集群。Kafka 通过 Zookeeper 管理集群配置，选举 leader，以及在 Consumer Group 发生变化时进行 rebalance。Producer 使用 push 模式将消息发布到 broker，Consumer 使用 pull 模式从 broker 订阅并消费消息。

2. ZooKeeper

ZooKeeper 是一个为分布式应用所设计的分布的、开源的协调服务。它提供了一些简单的操作，使得分布式应用可以基于这些接口实现诸如配置维护、域名服务、分布式同步、组服务等。ZooKeeper 的一些基本概念如表 6-3 所示。

表 6-3　ZooKeeper 的一些基本概念

角色		描述
领导者（leader）		领导者负责进行投票的发起和决议，更新系统状态
学习者（learner）	跟随者（follower）	Follower 用于接收客户端请求并向客户端发回结果，在选举过程中参与投票
	观察者(observer)	Observer 可以接收客户端连接，将写请求转发给 leader 节点。但 observer 不参加投票过程，只同步 leader 状态。Observer 目的是为了扩展系统，提高读取速度
客户端（client）		请求发起方

ZooKeeper 很容易编程接入，它使用了一个和文件树结构相似的数据模型。可以使用 Java 或者 C 来进行编程接入。ZooKeeper 系统模型如图 6-5 所示。

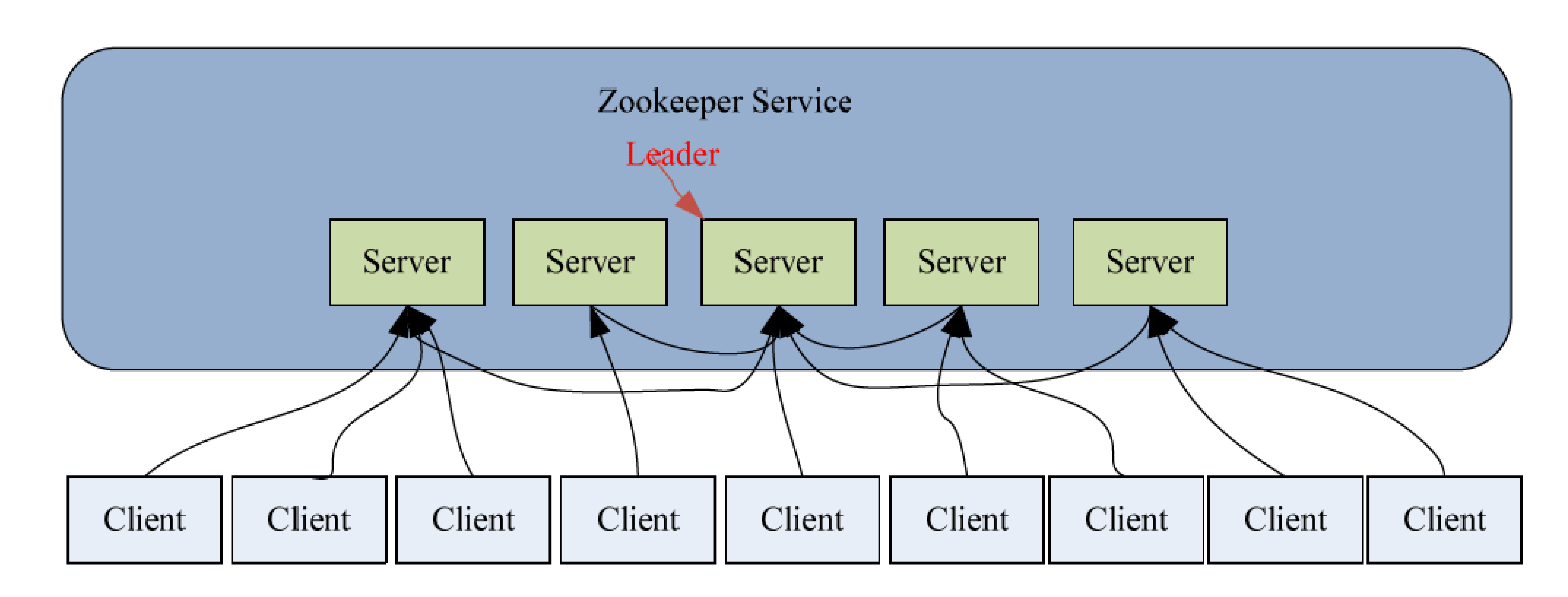


图 6-5　ZooKeeper 系统模型

ZooKeeper 保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。但由于网络延时等原因，ZooKeeper 不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用 sync()接口。

6.4.3　Spark 流式计算

对采集到的数据进行实时分析，选用 Apache 的 Spark，在 Spark2.x 中，Spark Streaming 获得了比较全面的升级，称为 Structured Streaming。

在 2.x 中，提出了一个叫做 continuous applications 连续应用程序的概念。如图 6-6 所示，数据从 Kafka 中流进来，通过 ETL 操作进行数据清洗，清洗出来作为目标数据进行进一步处理，可能是机器学习，也可能是交互式查询，也有可能直接把数据存在数据库或者其他外部存储设备，也有可能是直接交给已有的应用程序。也就是说 Spark Streaming 在获得数据后，能把全部处理环节串联起来，称之为端到端（End to end）处理。

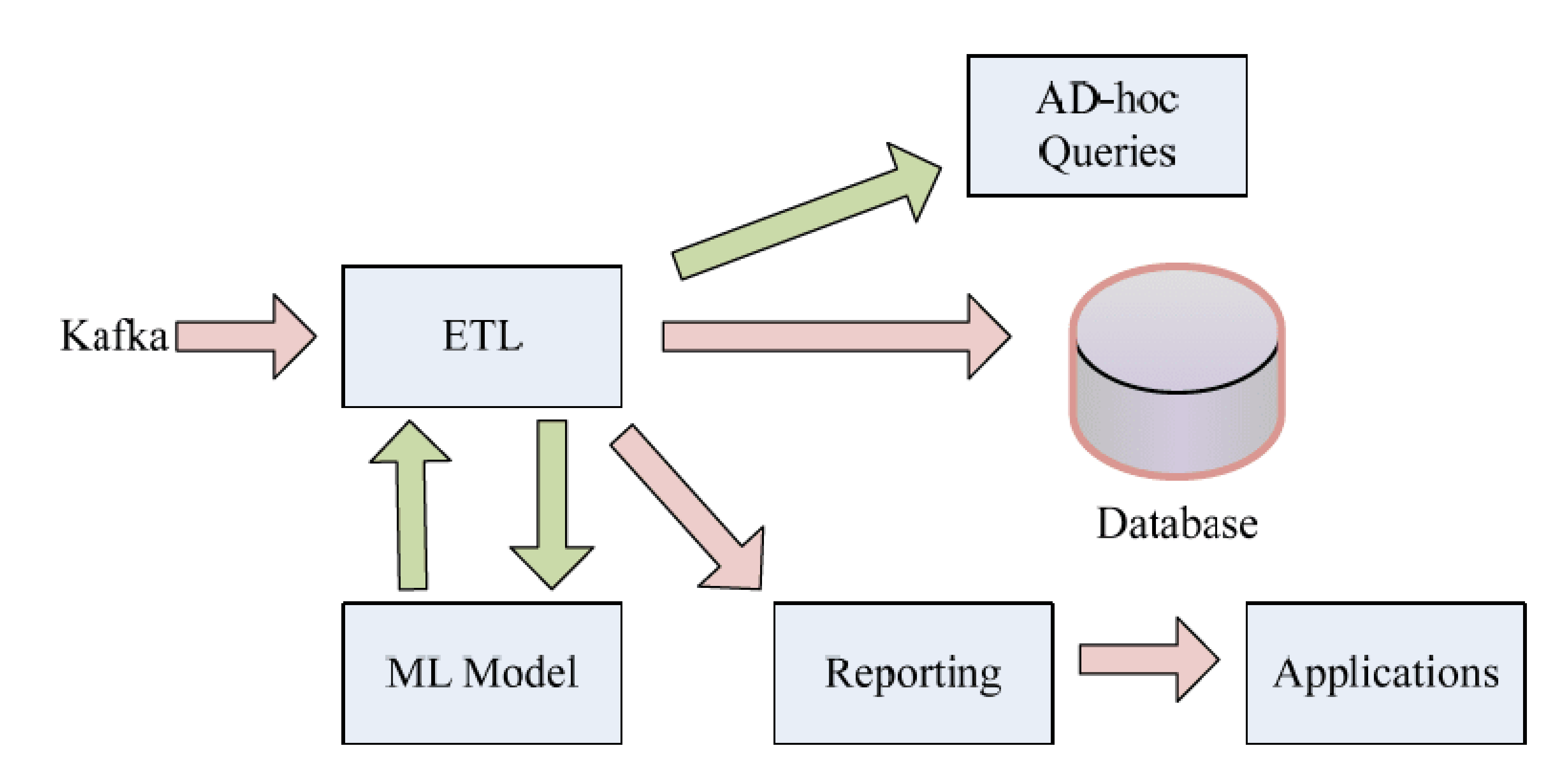


图 6-6　连续应用程序数据流处理

对 SparkStreaming 来说，Continuous 还有另一层含义，即运行在 DataSet 和 Dataframe 之上。基本观点是把数据看成一张表，默认情况下 DataSet 和 Dataframe 中的表是有边界的，而在流处理中是无边界的。对于 Spark Streaming 来说，是将数据抽象为了一个没有边界的表。

这个做法有一个非常大的好处，我们知道，目前 Spark Streaming 是直接依赖 RDD，优化需要自己完成，使用 DataSet 和 Dataframe 就可以利用 Tungsten 引擎来进行优化。把 Tungsten 等优化技术轻而易举地应用起来，可以说是在技术的运用上促进化学反应的发生。

在 API 方面，引入和流函数的封装。Kafka 中读取的数据，通过 stream 方法形成流，可以直接与 JDBC 中读取的数据在 DataSet 层面就进行 Join，不用使用 transform 或者 foreach RDD 方法。stream 方法底层依赖 Dataset 和 Dataframe，集成了 Spark SQL 和 Dataset 几乎所有的功能。

6.4.4 数据输出

对分析后的结果持久化，可以使用 HDFS、HBase、MongoDB。

HDFS 是一个主/从（Master/Slave）体系结构，从最终用户的角度来看，它就像传统的文件系统一样，可以通过目录路径对文件执行 CRUD（Create、Read、Update 和 Delete）操作。但由于分布式存储的性质，HDFS 集群拥有一个 NameNode 和一些 DataNode。NameNode 管理文件系统的元数据，DataNode 存储实际的数据。客户端通过与 NameNode 和 DataNodes 的交互访问文件系统。客户端联系 NameNode 以获取文件的元数据，而真正的文件 I/O 操作是直接和 DataNode 进行交互的。

HBase 是 Apache Hadoop 的数据库，能够对大型数据提供随机、实时的读写访问。HBase 的目标是存储并处理大型的数据，HBase 是一个开源的、分布式的、多版本的、面向列的存储模型，它存储的是松散型数据。HBase 是一个构建在 HDFS 上的分布式列存储系统。HBase 是基于 Google BigTable 模型开发的，是典型的 key/value 系统。HBase 是 Apache Hadoop 生态系统中的重要一员，主要用于海量结构化数据存储。从逻辑上讲，HBase 将数据按照表、行和列进行存储。与 Hadoop 一样，HBase 目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。

MongoDB 是一个基于分布式文件存储的数据库。由 C++语言编写，旨在为 Web 应用提供可扩展的高性能数据存储解决方案。MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富、最像关系数据库的。它支持的数据结构非常松散，是类似 JSON 的 BSON 格式，因此可以存储比较复杂的数据类型。MongoDB 最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

6.4.5 日志摄取

Web 日志包含着网站最重要的信息，通过日志分析，我们可以知道网站的访问量，哪个网页访问人数最多，哪个网页最有价值等。一般中型的网站（10W 的 PV 以上），每天会产生 1GB 以上 Web 日志文件。大型或超大型的网站，可能每小时就会产生 10GB 的数据量。

Web 日志由 Web 服务器产生，可能是 Nginx、Apache、Tomcat 等。从 Web 日志中，我

们可以获取网站每类页面的 PV 值（PageView，页面访问量）、独立 IP 数。稍微复杂一些的，可以计算得出用户所检索的关键词排行榜、用户停留时间最高的页面等。更复杂的，构建广告点击模型、分析用户行为特征等。

在 Web 日志中，每条日志通常代表着用户的一次访问行为。例如，下面就是一条 nginx 日志：

```
222.68.172.190 - - [18/Sep/2013:06:49:57 +0000] "GET /images/my.jpg HTTP/1.1"
200 19939
"http://www.angularjs.cn/A00n""Mozilla/5.0 (Windows NT 6.1)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/29.0.1547.66 Safari/537.36"
```

这条日志可以拆解为以下 8 个变量：

- remote_addr: 记录客户端的 ip 地址，222.68.172.190。
- remote_user: 记录客户端用户名称，–。
- time_local: 记录访问时间与时区，[18/Sep/2013:06:49:57 +0000]。
- request: 记录请求的 URL 与 HTTP 协议，“GET /images/my.jpg HTTP/1.1”。
- status: 记录请求状态，成功是 200，200。
- body_bytes_sent: 记录发送给客户端文件主体内容大小，19939。
- http_referer: 用来记录从那个页面链接访问过来的，http://www.angularjs.cn/A00n。
- http_user_agent: 记录客户浏览器的相关信息，Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/29.0.1547.66 Safari/537.36。

其他更多的信息，则要用其他手段去获取，通过 JS 代码单独发送请求，使用 Cookies 记录用户的访问信息。利用这些日志信息，我们可以深入挖掘网站的秘密。

1. 少量数据的情况

少量数据的情况（10MB、100MB、10GB），在单机处理尚能忍受的时候，我们可以直接利用各种 UNIX/Linux 工具，awk、grep、sort、join 等都是日志分析的利器，再配合 perl、Python、正则表达式等工具，基本就可以解决所有的问题。

2. 海量数据的情况

当数据量每天以 10GB、100GB 增长的时候，单机处理能力已经不能满足需求。我们就需要增加系统的复杂性，用计算机集群、存储阵列来解决。在 Hadoop 出现之前，海量数据存储和海量日志分析都是非常困难的。只有少数一些公司，掌握着高效的并行计算、分步式计算、分步式存储的核心技术。

3. 日志摄取

日志摄取应用被用作消费应用日志，例如 Web 访问日志。为了简化使用场景，提供一个 Web 访问日志，模拟访客浏览产品目录，这些日志代表了单击流日志，既用作长时处理，也用作实时推荐。架构有两个选择：第一个是以 Flume 来传输日志；第二个是以 ELK 来创建

访问分析。

ELK 是 Elasticsearch、Logstash、Kibana 的简称，这三者是核心套件，但并非全部。

- Elasticsearch 是实时全文搜索和分析引擎，提供搜集、分析、存储数据三大功能。它是一套开放 REST 和 JAVA API 等结构，提供高效搜索功能，可扩展的分布式系统。它构建于 Apache Lucene 搜索引擎库之上。
- Logstash 是一个用来搜集、分析、过滤日志的工具。它支持几乎任何类型的日志，包括系统日志、错误日志和自定义应用程序日志。它可以从许多来源接收日志，这些来源包括 syslog、消息传递（例如 RabbitMQ）和 JMX，它能够以多种方式输出数据，包括电子邮件、websockets 和 Elasticsearch。
- Kibana 是一个基于 Web 的图形界面，用于搜索、分析和可视化存储在 Elasticsearch 中的日志数据。它利用 Elasticsearch 的 REST 接口来检索数据，不仅允许用户创建他们自己的数据的定制仪表板视图，还允许他们以特殊的方式查询和过滤数据。

ELK 的一种架构如图 6-7 所示。

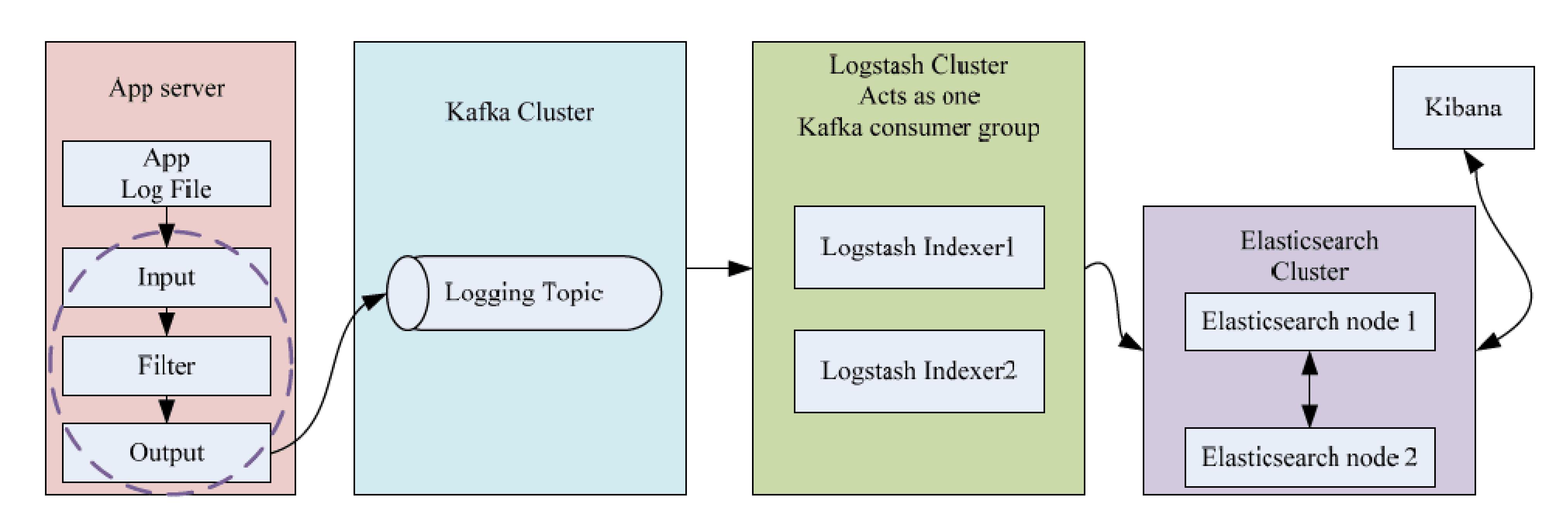


图 6-7　ELK 一种架构

架构引入了消息队列机制，位于各个节点上的 Logstash Agent 先将数据/日志传递给 Kafka（或者 Redis），并将队列中消息或数据间接传递给 Logstash，Logstash 过滤、分析后将数据传递给 Elasticsearch 存储。最后由 Kibana 将日志和数据呈现给用户。因为引入了 Kafka（或者 Redis），所以即使远端 Logstash Server 因故障停止运行，数据将会先被存储下来，从而避免数据丢失。

图 6-8 展示了 ELK 和 Flume 是如何处理日志的。

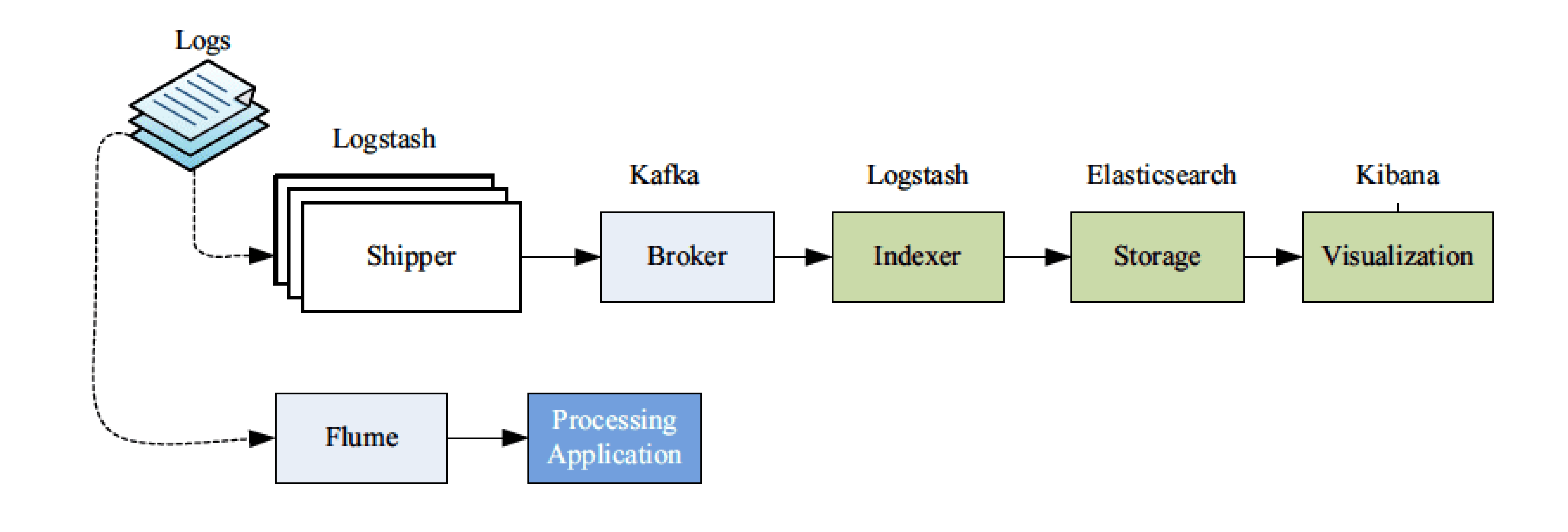


图 6-8　ELK 结合 Flume 处理日志

在架构中使用 ELK ，因为 LEK 的三个产品无缝集成，能够比使用 Flume 给我们更多的价值。

4. KPI 指标设计

在进行海量 Web 日志分析，提取 KPI 数据时，KPI 指标设计如下：

- PV（PageView）：页面访问量统计。
- IP: 页面独立 IP 的访问量统计。
- Time: 用户每小时 PV 的统计。
- Source: 用户来源域名的统计。
- Browser: 用户的访问设备统计。

6.4.6　机器学习

机器学习应用接收数据流，构建推荐引擎。应用使用一个基本的算法来基于 Spark MLlib 介绍机器学习的概念（如第 5 章的 Spark MLlib）。

图 6-9 展示了该机器学习应用如何使用 Kafka 接收数据，然后发送给 Spark 处理，最后在 Elasticsearch 中建立索引为将来使用做准备。

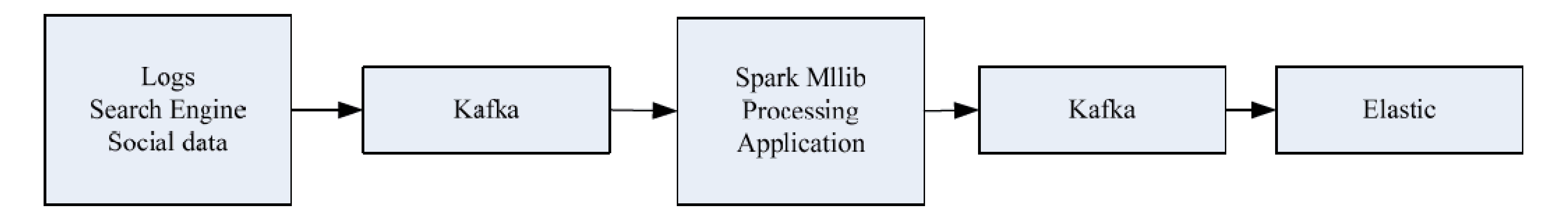


图 6-9　Spark MLlib 使用 Kafka 接收数据

6.4.7　处理引擎

处理引擎是大数据处理架构的心脏，它接收各种源的数据，代理合适模型的处理。图 6-10 展示了由 Hive 组成的处理引擎如何接收数据，以及 Spark 的实时/准实时处理。

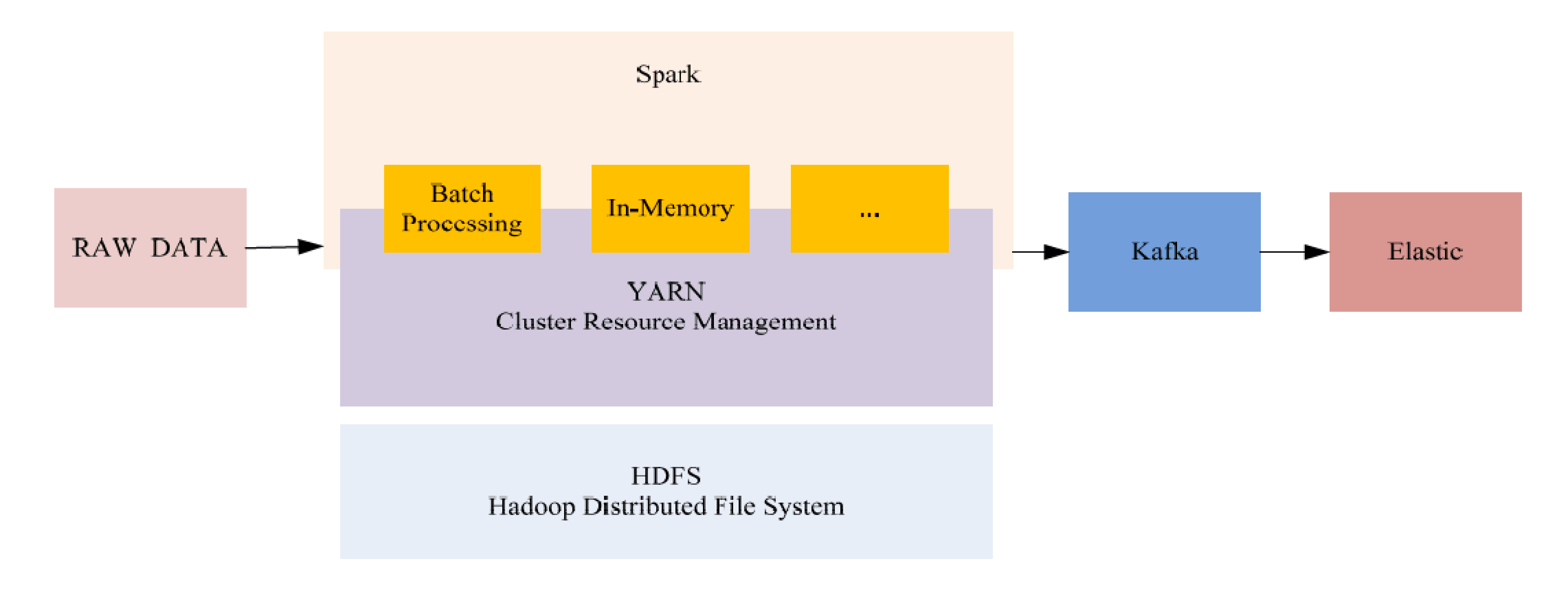


图 6-10　由 Hive 组成的处理引擎接收数据

这里使用 Kafka 与 Logstash 结合把数据分发给 Elasticsearch。Spark 位于 Hadoop 集群的顶端，但不是必需的。为了简化起见，可以不建立 Hadoop 集群，而是以 standalone 模式运行 Spark。显然，应用同样可以部署在所选择的 Hadoop 发布版上。

6.5 Spark 单个机器集群部署

除了在 Mesos 或 YARN 集群上运行之外，Spark 还提供一个简单的独立部署的模块。你可以通过手动开始 Master 和 Workers 来启动一个独立的集群；你也可以利用我们提供的脚本，运行这些进程在单个机器上进行测试。

1. 安装 Spark 独立集群

部署 Spark 最简单的方法就是运行./make-distribution.sh 脚本来生成一个二进制发行版。这个版本能部署在任意运行 Java 的机子上，不需要安装 Scala。

建议的步聚是先在一个节点部署并启动 Master，获得 master spark URL，在 dist/这个目录下修改 conf/spark-env.sh，然后再部署到其他的节点上。

2. 手动启动集群

通过如下命令启动单独模式的 Master 服务：

```
./bin/start-master.sh
```

一旦启动，Master 就会输出 spark://IP:PORT，以提示连接 Workers 的方式。也可以通过参数“master”给 SparkContext 来连接集群的作业。你可以在 Master 的 Web 管理界面上看到这样的地址，默认是 http://localhost:8080。

同样，你可以启动一个或者多个 Worker，通过下面的语句使之和 Master 建立连接：

```
./spark-class org.apache.spark.deploy.worker.Worker spark://IP:PORT
```

启动一个 Worker 后，查看 Master 的 Web 管理界面（默认 http://localhost:8080），上面列出了新近加入的节点的 CPU 和内存的信息（不包括给操作系统预留的内存空间），如表 6-4 所示。

表 6-4 Master 和 Worker 的一些配置选项

参数	含义
-i IP,--ip IP	要监听的 IP 地址或者 DNS 机器名
-p PORT,--port PORT	要监听的端口，默认：master 7077。Worker 随机
--webui-port PORT	Web UI 端口，默认：master 8080。Worker 8081
-c CORES,--cores CORES	作业可用的 CPU 内核数量，默认：所有可用。只在 Worker 上
-m MEM,--memory MEM	作业可使用的内存容量，默认格式 1000MB 或者 2GB，默认：所有 RAM 去掉给操作系统用的 1GB。只在 Worker 上
-d DIR,--work-dir DIR	伸缩空间和日志输入的目录路径，默认：SPARK_HOME/work。只在 Worker 上

3. 集群启动脚本

通过脚本启动 Spark 独立集群时，需要在 Spark 目录下创建一个文件 conf/slaves，列出所有启动的 Spark workers 的主机名，每行一条记录。Master 必须能够实现通过 ssh（使用私钥）访问 Worker 机器，可以使用 ssh localhost 来测试。

一旦建立了这个档案，可以通过以下脚本停止或启动集群，这些脚本基于 Hadoop 的部署脚本，在 SPARK_HOME/bin 目录：

- bin/start-master.sh：在机器上执行脚本，启动 Master 。
- bin/start-slaves.sh：启动 conf/slaves 中指定的每一个 Slave 。
- bin/start-all.sh：同时启动 Master 以及上面文件中指定的 Slave。
- bin/stop-master.sh：停止通过 bin/start-master.sh 脚本启动的 Master。
- bin/stop-slaves.sh：停止通过 bin/start-slaves.sh 启动的 Slave。
- bin/stop-all.sh：停止上述的两种启动脚本启动的 Master 和 Slave。

注意：只能在运行 Spark 的 Master 主机上执行上述脚本，而不是你的本地机器。

可以通过 conf/spark-env.sh 进一步配置整个集群的环境变量。这个文件可以用 conf/spark-env.sh.template 当模版复制生成。然后，复制到所有的 Worker 机器上才奏效。表 6-5 给出一些可选的参数以及含义。

表 6-5　配置集群一些可选的参数以及含义

环境变量	含义
SPARK_MASTER_IP	绑定一个外部 IP 给 Master
SPARK_MASTER_PORT	从另外一个端口启动 Master，默认：7077
SPARK_MASTER_WEBUI_PORT	Master 的 Web UI 端口，默认：8080
SPARK_WORKER_PORT	启动 Spark worker 的专用端口，默认：随机
SPARK_WORKER_DIR	伸缩空间和日志输入的目录路径，默认：SPARK_HOME/work
SPARK_WORKER_CORES	作业可用的 CPU 内核数量，默认：所有可用的
SPARK_WORKER_MEMORY	作业可使用的内存容量，默认格式 1000MB 或者 2GB，默认：所有 RAM 去掉给操作系统用的 1 GB。注意：每个作业自己的内存空间由 SPARK_MEM 决定
SPARK_WORKER_WEBUI_PORT	Worker 的 Web UI 启动端口，默认：8081
SPARK_WORKER_INSTANCES	没在机器上运行 Worker 数量，默认:1。当你有一个非常强大的计算机的时候和需要多个 Spark worker 进程的时候你可以修改这个默认值大于 1。如果你设置了这个值。要确保 SPARK_WORKER_CORE 明确限制每一个 r worker 的核心数，否则每个 Worker 将尝试使用所有的核心
SPARK_DAEMON_MEMORY	分配给 Spark master 和 Worker 守护进程的内存空间 ，默认：512MB
SPARK_DAEMON_JAVA_OPTS	Spark master 和 Worker 守护进程的 JVM 选项，默认：none



启动脚本目前不支持 Windows。要在 Windows 上运行一个 Spark 集群，需要手动启动 Master 和 Workers。

4. 集群连接应用程序

在 Spark 集群上运行一个应用，只需通过 Master 的 spark://IP:PORT 链接传递到 SparkContext 构造器。

在集群上运行交互式的 Spark 命令，运行如下命令：

```
MASTER=spark://IP:PORT ./spark-shell
```

注意，如果你在一个 Spark 集群上运行了 spark-shell 脚本，spark-shell 将通过在 conf/spark-env.sh 下的 SPARK_MASTER_IP 和 SPARK_MASTER_PORT 自动设置 Master 。

也可以传递一个参数-c <numCores> 来控制 spark-shell 在集群上使用的核心数量。

5. 资源调度

单独部署模式目前只支持 FIFO 作业调度策略。不过，为了允许多并发执行，你可以控制每一个应用可获得资源的最大值。默认情况下，如果系统中只运行一个应用，它就会获得所有资源。使用类似 System.setProperty(“spark.cores.max”,“10”)的语句可以获得内核的数量。这个数值在初始化 SparkContext 之前必须设置好。

6. 监控和日志

Spark 单独部署模式提供了一个基于 Web 的集群监视器。Master 和每一个 Worker 都会有一个 Web UI 来显示集群的统计信息。默认情况下，可以通过 8080 端口访问 Master 的 Web UI。当然也可以通过配置文件或者命令来修改这个端口值。

另外，每个 Slave 节点上作业运行的日志也会详细地记录到默认的 SPARK_HOME/work 目录下。每个作业会对应两个文件：stdout 和 stderr，包含了控制台上的所有的历史输出。

7. 和 Hadoop 同时运行

Spark 作为一个独立的服务，可以和现有的 Hadoop 集群同时运行。 通过 hdfs:// URL，Spark 可以访问 Hadoop 集群的 HDFS 上的数据。比如，地址可以写成 hdfs://<namenode>:9000/path，从 Namenode 的 Web UI 可以获得更确切的 URL。或者，专门为 Spark 搭建一个集群，通过网络访问其他 HDFS 上的数据，这样肯定不如访问本地数据速度快，除非是都在同一个局域网内。比如，几台 Spark 机器和 Hadoop 集群在同一机架上。

6.6 本章小结

将大数据架构分成三个部分：批处理、流处理和服务架构。当处理海量数据的时候，Spark 带来了大量的解决方案，但也为资源分配和管理存储数据带来了挑战，我们总是希望在保持最小时延的同时而降低成本。当处理海量成分混杂数据的时候，社交网络是复杂性的代表。除了数据结构，还需要将数据分类成逻辑上的子集以便增强数据处理的效果。考虑情绪分析的例子，从大数据集的非结构化数据中得到有价值信息的位置来组成数据。大量的 IT

组织如今都有自己的数据架构，因为都依赖于传统的数据架构，处理多数据源已不再新鲜，这些架构已经连接了多维度的数据源，例如 CRM 系统、文件系统和其他商用系统。主要运行的关系型数据库有 Oracle、DB2 和 Microsoft SQL。

本章从典型商务使用场景出发，简要介绍了 Spark 的三种分布式部署模式，然后详细介绍了创建大数据架构的组成元素与流程，最后以 Spark 单机集群部署为例介绍了集群部署问题与步骤。

第 7 章

Spark大数据处理案例分析

Spark 作为 Apache 顶级的开源项目，项目主页参见 http://spark.apache.org。在迭代计算、交互式查询计算以及批量流计算方面都有相关的子项目，如 Shark、Spark Streaming、MLbase、GraphX、SparkR 等。

随着企业数据量的增长，对大数据的处理和分析已经成为企业的迫切需求。Spark 作为 Hadoop 的替代者，引起学术界和工业界的普遍兴趣，大量应用在工业界落地，许多科研院校开始了对 Spark 的研究。涉及 Benchmark、SQL、并行算法、性能优化、高可用性等多个方面。

互联网用户群体庞大，需要存储大数据并进行数据分析，Spark 能够支持多范式的数据分析，解决了大数据分析中迫在眉睫的问题。例如，国外 Cloudera、MapR 等大数据厂商全面支持 Spark，微策略等老牌 BI 厂商也和 Databricks 达成合作关系，Yahoo!使用 Spark 进行日志分析并积极回馈社区，Amazon 在云端使用 Spark 进行分析。国内同样得到很多公司的青睐，淘宝构建 Spark on Yarn 进行用户交易数据分析，使用 GraphX 进行图谱分析。网易用 Spark 和 Shark 对海量数据进行报表和查询。腾讯使用 Spark 进行精准广告推荐。下面将选取代表性的 Spark 应用案例进行分析[101]。

7.1 Spark on Amazon EMR

7.1.1 Amazon EMR

利用 Amazon EMR，你可以分析和处理大量数据。它通过在 Amazon 云上运行的虚拟服务器集群中分配计算工作来实现此目的。使用名为 Hadoop 的开源框架管理该集群。

Hadoop 使用称为 MapReduce 的分布式处理架构，其中任务被映射到一组服务器以供处理。然后，由这些服务器执行的计算结果将减少为单个输出集。其中一个节点被指定为主节点，它控制任务的分配[102]。

Amazon EMR 增强了 Hadoop 和其他开源应用程序，以便与 AWS 无缝协作。例如，在 Amazon EMR 上运行的 Hadoop 集群，使用 EC2 实例作为虚拟 Linux 服务器用于主节点和从属节点，将 Amazon S3 用于输入和输出数据的批量存储，并将 CloudWatch 用于监

控集群性能和发出警报。你还可以使用 Amazon EMR 和 Hive 将数据迁移到 DynamoDB 以及从中迁出。所有这些操作都由启动和管理 Hadoop 集群的 Amazon EMR 控制软件进行编排。这个流程名为 Amazon EMR 集群。如图 7-1 所示为 Amazon EMR 如何与其他 AWS 服务交互[103]。

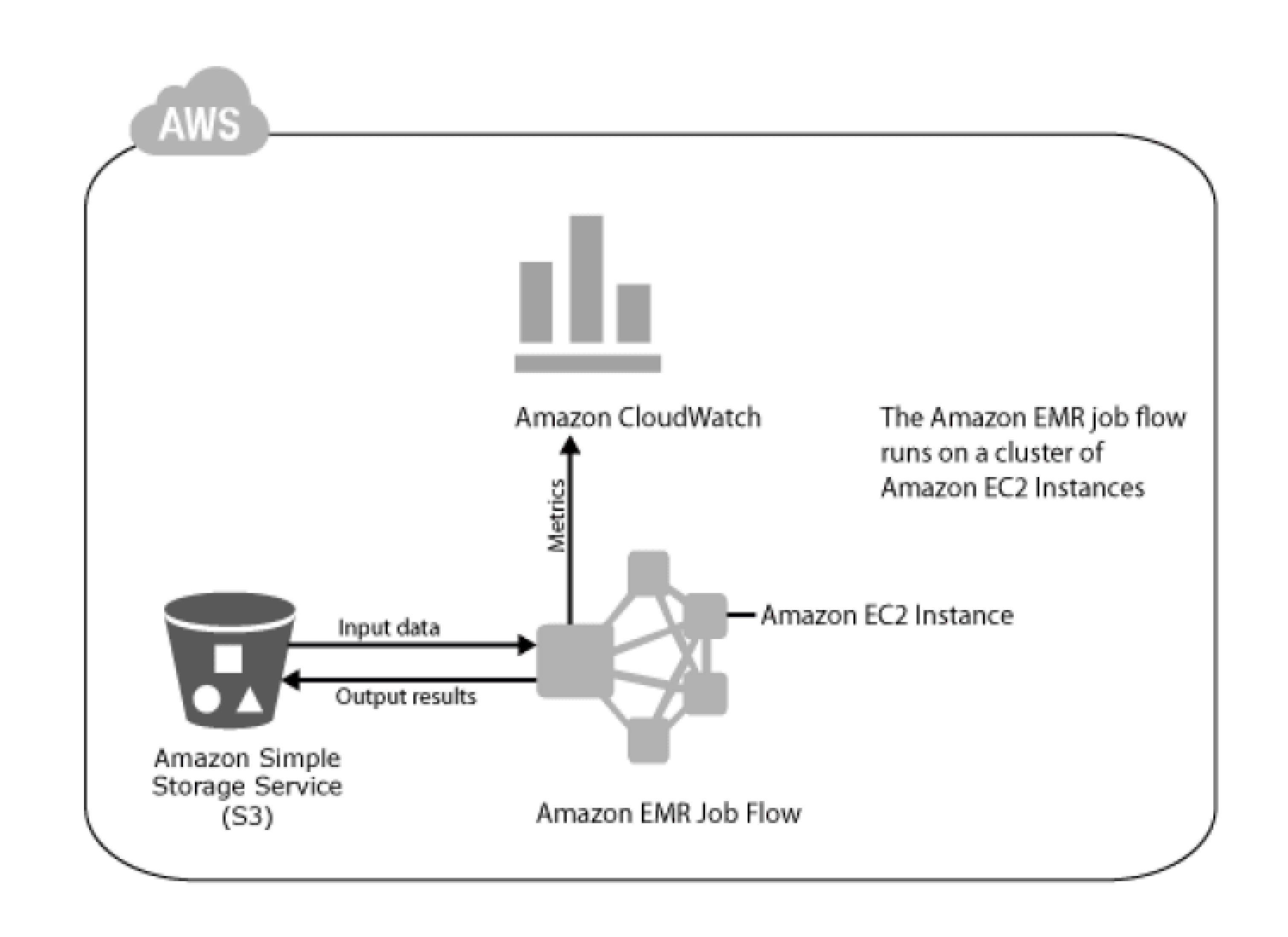


图 7-1　Amazon EMR 如何与其他 AWS 服务交互

Amazon EMR 本身支持 Hadoop YARN 上的 Apache Spark，可以从 AWS 管理控制台、AWS CLI 或 Amazon EMR API 轻松快速地创建托管的 Apache Spark 集群。此外，你还可以利用其他 Amazon EMR 功能，包括使用 Amazon EMR 文件系统（EMRFS）快速连接 Amazon S3、与 Amazon EC2 竞价型市场集成，以及使用 Auto Scaling 在集群中添加或移除实例。借助 Apache Spark，你还可以使用 Apache Zeppelin 创建交互式和协作式笔记本电脑以进行数据探索。

借助 Amazon EMR Step API 提交 Apache Spark 作业，结合使用 Apache Spark 和 EMRFS 以直接访问 Amazon S3 中的数据，使用 Amazon EC2 竞价型容量节约成本，使用 Auto Scaling 动态添加和移除容量，并根据你的工作负载启动长期运行的集群或临时集群。你还可以使用 Amazon EMR 安全、配置 Spark 加密。Amazon EMR 在 Hadoop YARN 上安装和管理 Apache Spark，你还可以在集群中添加其他 Hadoop 生态系统应用程序。

7.1.2　配置 Spark

要配置 Spark，你需要在创建集群时通过提供参数的方式进行。根据位于 GitHub 的引导操作，配置 Amazon EMR 上的 Spark。CLI 和控制台将接收此处记录的选项作为参数。你可以配置 Apache Spark 文档给出的 Spark 配置主题中列出的任意选项，你可以在 $SPARK_CONF_DIR/spark-defaults.conf 配置文件中查看现有集群的这些配置。在创建集群时提供以下参数：

- -d key=value：提供 SparkConf 设置以覆盖 spark-defaults.conf。要指定多个选项，请在每个键值对前插入 -d。
- -c S3_Path：存储在 Amazon S3 上的 spark-install 配置文件的位置。
- -g：为 Spark 安装 Ganglia 指标配置。
- -a：将 spark-assembly-*.jar 放在 Spark 类路径中所有系统 JAR 的前面。
- -u S3_Path：将给定 S3 URI 处的 JAR 添加到 Spark 类路径中。此选项的优先级高于 -a 选项，因此，如果与 -a 一同指定，添加的 JAR 将优先于 spark-assembly-*.jar。
- -l logging_level：为驱动程序的 log4j.logger.org.apache.spark 设置日志记录级别。选项有 OFF、ERROR、WARN、INFO、DEBUG 或 ALL。默认为 INFO。
- -h：使用 Amazon EMR 自定义 Hive JAR 代替 Spark 提供的 JAR。
- -x：为专用 Spark 作业利用率设置默认 Spark 配置（例如：每个节点、所有虚拟核心和内存、所有核心节点、1 个执行者）。

使用下面的命令创建一个安装了 Spark，且 spark.executor.memory 设为 2GB 的集群：

```
aws emr create-cluster --name "Spark cluster" --ami-version 3.11.0 --
applications Name=Spark,\

Args=[-d,spark.executor.memory=2G] --ec2-attributes KeyName=myKey --instance-
type m3.xlarge --instance-count 3 --use-default-roles
```

使用控制台创建一个 spark.executor.memory 设为 2GB 的集群：

（1）通过以下网址打开 Amazon EMR 控制台：https://console.aws.amazon.com/elasticmapreduce/。

（2）选择 Create cluster。

（3）对于 Software Configuration 字段，选择 Amazon AMI Version 3.9.0 或更高版本。

（4）对于 Applications to be installed 字段，从列表中选择 Spark，然后选择 Configure and add，然后将参数 spark.executor.memory=2G 放入参数框内并选择 Add。

（5）根据需要选择其他选项，然后选择 Create cluster。

7.1.3 以交互方式或批处理模式使用 Spark

Spark 内在支持用 Scala、Python 和 Java 编写的应用程序，包含几个用于 SQL（Spark SQL）、机器学习（MLlib）、流式处理（Spark Streaming）和图形处理（GraphX）的紧密集成库。这些工具可让你更轻松地在各种使用案例中充分发挥 Spark 框架的优势。

Spark 可与 Amazon EMR 中可用的其他 Hadoop 应用程序一同安装，而且，它还能借助 EMR 文件系统（EMRFS）直接访问 Amazon S3 中的数据。此外，Hive 也与 Spark 集成。因此，我们可以通过 HiveContext 对象运行使用 Spark 的 Hive 脚本。Hive 上下文作为 sqlContext 包含在 Spark 外壳中。

Amazon EMR 以两种模式运行 Spark 应用程序：

- 交互式
- 批处理

当我们使用控制台或 AWS CLI 启动长时间运行的集群时，可以通过 SSH 以 Hadoop 用户身份连接到主节点，使用 Spark 外壳以交互方式开发并运行 Spark 应用程序。与批处理环境相比，以交互方式使用 Spark 能够让我们更轻松地对 Spark 应用程序进行原型设计或测试。在交互模式下成功修改 Spark 应用程序后，可以将该应用程序 JAR 或 Python 程序放到 Amazon S3 上集群主节点的本地文件系统上。然后，将应用程序作为批处理工作流程提交。

在批处理模式中，将 Spark 脚本上传到 Amazon S3 或本地主节点文件系统，然后将此工作作为步骤提交到集群。Spark 步骤可提交到长时间运行的集群或暂时性集群。

7.1.4　使用 Spark 创建集群

按照 7.1.2 节过程创建一个安装了 Spark 的集群。使用 AWS CLI 启动安装了 Spark 的集群。

使用下面的命令创建集群：

```
aws emr create-cluster --name "Spark cluster" --ami-version 3.10.0 --
applications Name=Spark \
--ec2-attributes KeyName=myKey --instance-type m3.xlarge --instance-count 3 --
use-default-roles
```

注意：对于 Windows，将上述 Linux 行继续符（\）替换为脱字符（^）。

通过 RunJobFlowRequest 中使用的 SupportedProductConfig 指定 Spark 作为应用程序。

下面的 Java 程序片段显示如何使用 Spark 创建集群：

```
AmazonElasticMapReduceClient emr = new
AmazonElasticMapReduceClient(credentials);
SupportedProductConfig sparkConfig = new SupportedProductConfig()
            .withName("Spark");

RunJobFlowRequest request = new RunJobFlowRequest()
            .withName("Spark Cluster")
            .withAmiVersion("3.11.0")
            .withNewSupportedProducts(sparkConfig)
            .withInstances(new JobFlowInstancesConfig()
                .withEc2KeyName("myKeyName")
                .withInstanceCount(1)
                .withKeepJobFlowAliveWhenNoSteps(true)
                .withMasterInstanceType("m3.xlarge")
                .withSlaveInstanceType("m3.xlarge")
            );
RunJobFlowResult result = emr.runJobFlow(request);
```

7.1.5 访问 Spark 外壳

Spark 外壳基于 Scala REPL（读取-求值-输出-循环）。它让你能够以交互方式创建 Spark 程序并将工作提交到框架。可以通过 SSH 连接主节点并调用 spark-shell，从而访问 Spark shell（外壳）。

默认情况下，Spark shell 创建其自己的 SparkContext 对象（称作 sc）。如果 REPL 中需要，可以使用此上下文。sqlContext 也可在此外壳中使用，它是一种 HiveContext。

可以使用 Spark shell 统计存储在 Amazon S3 上某个文件中的某个字符串的出现次数。比如下面的示例，使用 sc 读取 Amazon S3 中的 textFile。

```
scala> sc
res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@404721db

scala> val textFile = sc.textFile("s3://elasticmapreduce/samples/hive-
ads/tables/impressions/dt=2009-04-13-08-05/ec2-0-51-75-39.amazon.com-2009-04-
13-08-05.log")
```

Spark 创建 textFile 及关联的数据结构。然后，上面示例会统计此日志文件中包含字符串“cartoonnetwork.com”的行数：

```
scala> val linesWithCartoonNetwork = textFile.filter(line =>
line.contains("cartoonnetwork.com")).count()
linesWithCartoonNetwork: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2]
at filter at <console>:23
<snip>
<Spark program runs>
scala> linesWithCartoonNetwork
res2: Long = 9
```

Spark 还包含一个基于 Python 的外壳 pyspark，你可以用它来设计以 Python 编写的 Spark 程序的原型。与使用 spark-shell 的方法一样，在主节点上调用 pyspark 即可；它包含同样的 SparkContext 对象。

```
>>> sc
<pyspark.context.SparkContext object at 0x7fe7e659fa50>
>>> textfile = sc.textFile("s3://elasticmapreduce/samples/hive-
ads/tables/impressions/dt=2009-04-13-08-05/ec2-0-51-75-39.amazon.com-2009-04-
13-08-05.log")
```

Spark 创建 textFile 及关联的数据结构。然后，示例会统计此日志文件中包含字符串“cartoonnetwork.com”的行数。

```
>>> linesWithCartoonNetwork = textfile.filter(lambda line:
```

```
"cartoonnetwork.com" in line).count()
15/06/04 17:12:22 INFO lzo.GPLNativeCodeLoader: Loaded native gpl library from
the embedded binaries
15/06/04 17:12:22 INFO lzo.LzoCodec: Successfully loaded & initialized native-
lzo library [hadoop-lzo rev EXAMPLE]
15/06/04 17:12:23 INFO fs.EmrFileSystem: Consistency disabled, using
com.amazon.ws.emr.hadoop.fs.s3n.S3NativeFileSystem as filesystem
implementation
<snip>
<Spark program continues>
>>> linesWithCartoonNetwork
9
```

7.1.6 添加 Spark

可以使用 Amazon EMR 步骤向安装在 EMR 集群上的 Spark 框架提交工作。在控制台和 CLI 中，使用 Spark 应用程序，借助 API，通过 script-runner.jar 调用 spark-submit。如果选择使用客户端部署模式向 Spark 部署工作，应用程序文件必须位于 EMR 集群上的本地路径中。

使用控制台提交 Spark 步骤说明如下：

（1）通过以下网址打开 Amazon EMR 控制台：https://console.aws.amazon.com/elasticmapreduce/。
（2）在 Cluster List 中，选择你的集群的名称。
（3）滚动到 Steps 部分并展开它，然后选择 Add step。
（4）在 Add Step 对话框中按顺序做如下操作：

- 对于 Step type，选择 Spark application。
- 对于 Name，接受默认名称 （Spark application）或键入新名称。
- 对于 Deploy mode，选择 Cluster 或 Client 模式。集群模式在集群上启动你的驱动程序（对于基于 JVM 的程序，此为 main()），而客户端模式本地启动驱动程序。
- 指定所需的 Spark-submit options。
- 对于 Application location，指定应用程序的本地或 S3 URI 路径。
- 对于 Arguments，将该字段保留为空白。
- 对于 Action on failure，接受默认选项 （Continue）。

集群模式能够使用 S3 URI 提交工作。客户端模式要求将应用程序放到集群主节点的本地文件系统上。

（5）选择 Add。步骤会出现在控制台中，其状态为“Pending”。

（6）步骤的状态会随着步骤的运行从 Pending 变为 Running，再变为 Completed。要更新状态，选择 Actions 列上方的 Refresh 图标。

（7）如果配置了日志记录，则这一步的结果将放在 Amazon EMR 控制台的“Cluster Details”页面上的步骤旁边的 Log Files 下。在启动集群时，可以选择在配置的日志存储桶

中查找步骤信息。

在创建集群时提交步骤，或使用 aws emr add-steps 子命令在现有集群中提交步骤。

1. 使用 create-cluster

（1）Linux、UNIX 和 Mac OS X 用户：

```
aws emr create-cluster --name "Add Spark Step Cluster" --ami-version 3.11.0 --
applications Name=Spark\
 --ec2-attributes KeyName=myKey --instance-type m3.xlarge --instance-count 3 \
 --steps Type=Spark,Name="Spark Program",ActionOnFailure=CONTINUE,Args=[--
class,org.apache.spark.examples.SparkPi,/home/hadoop/spark/lib/spark-examples-
1.3.1-hadoop2.4.0.jar,10]
```

（2）Windows 用户：

```
aws emr create-cluster --name "Add Spark Step Cluster" --ami-version 3.11.0 --
applications Name=Spark --ec2-attributes KeyName=myKey --instance-type
m3.xlarge --instance-count 3 --steps Type=Spark,Name="Spark
Program",ActionOnFailure=CONTINUE,Args=[--
class,org.apache.spark.examples.SparkPi,/home/hadoop/spark/lib/spark-examples-
1.3.1-hadoop2.4.0.jar,10]
```

对于 Windows，将上述 Linux 行继续符（\）替换为脱字符（^）。

2. 向正在运行的集群添加步骤，使用 add-steps

（1）Linux、UNIX 和 Mac OS X 用户：

```
aws emr add-steps --cluster-id j-2AXXXXXXGAPLF --steps Type=Spark,Name="Spark
    Program",\         Args=[--
    class,org.apache.spark.examples.SparkPi,/home/hadoop/spark/lib/spark-examples-*.jar,10]
```

（2）Windows 用户：

```
aws emr add-steps --cluster-id j-2AXXXXXXGAPLF --steps Type=Spark,Name="Spark
    Program", Args=[--
    class,org.apache.spark.examples.SparkPi,/home/hadoop/spark/lib/spark-
    *.jar,10]
```

根据需要为不同的应用程序设置 Spark 默认配置值。可以在提交应用程序时使用步骤完成此操作（实质上是向 spark-submit 传递选项）。例如，可以通过更改 spark.executor.memory 来更改为执行者进程分配的内存，可以为 --executor-memory 开关提供类似下面的参数：

```
/home/hadoop/spark/bin/spark-submit --executor-memory 1g -
```

```
    classorg.apache.spark.examples.SparkPi
/home/hadoop/spark/lib/spark-examples*.jar 10
```

同样地，也可以调节 --executor-cores 和 --driver-memory。在步骤中，可以向步骤提供以下参数：

```
--executor-memory 1g --class org.apache.spark.examples.SparkPi
    /home/hadoop/spark/lib/ spark-
examples*.jar 10
```

还可以使用 --conf 选项调节没有内置开关的设置。

7.2 Spark 在 AWSKrux 的应用

作为用于管理客户信息的数据管理平台的一部分，Krux 使用 Apache Spark 运行许多机器学习和常规处理工作负载。Krux 结合使用临时 Amazon EMR 集群和 Amazon EC2 竞价型容量来节约成本，并将 Amazon S3 与 EMRFS 用作 Apache Spark 的数据层[104]。

Krux 的基础设施一直记录着每一个客户的数字足迹，包括网站、设备、应用程序和活动，Krux 是嵌入在客户和消费者之间的每一个数字互动。至关重要的是给消费者提供快速的性能、无限的缩放能力、实时的个性化体验，以及数据驱动。

Krux 收集、存储，使观众每一片段数据不断提供给其客户，管理超过 10 PB 的按需数据。全面观察媒体的行为，不只是运动，减少偏见，并为客户提供更高的精度和更复杂的细分定位和分析。Krux 不需要客户使用一个预先确定的受众分类；相反，它使他们能够独立地改变他们的受众分类需要，让他们能够迅速适应不断变化的业务需求。

Krux 提供 AWS 管理数据处理的要求，覆盖多种方式，包括准实时、按需和批处理模式执行 PB 级的数据需求分析。Krux 组合使用 Amazon EMR 和 Apache Spark 来运行机器学习和数据提取/转换/加载（ETL）的工作，与亚马逊简单存储服务（Amazon S3）为核心的分布式存储系统协同。使用 Amazon EC2 Spot 实例 Krux 实现了亚马逊的 EMR 基础设施来实现降低访问计算成本的功能，使用内部框架确定现货报价。

Krux 使用 AWS 数据管道服务功能，调度 AWS 的 Apache Hadoop 和 Apache Spark 的工作服务移动数据计算和存储。Krux 使用 Amazon EMR 管理按需批处理和数据处理科学框架，同时采用 Amazon DynamoDB 存储来自不同设备和应用的全球会员数据。krux 大数据解决方案如图 7-2 所示。

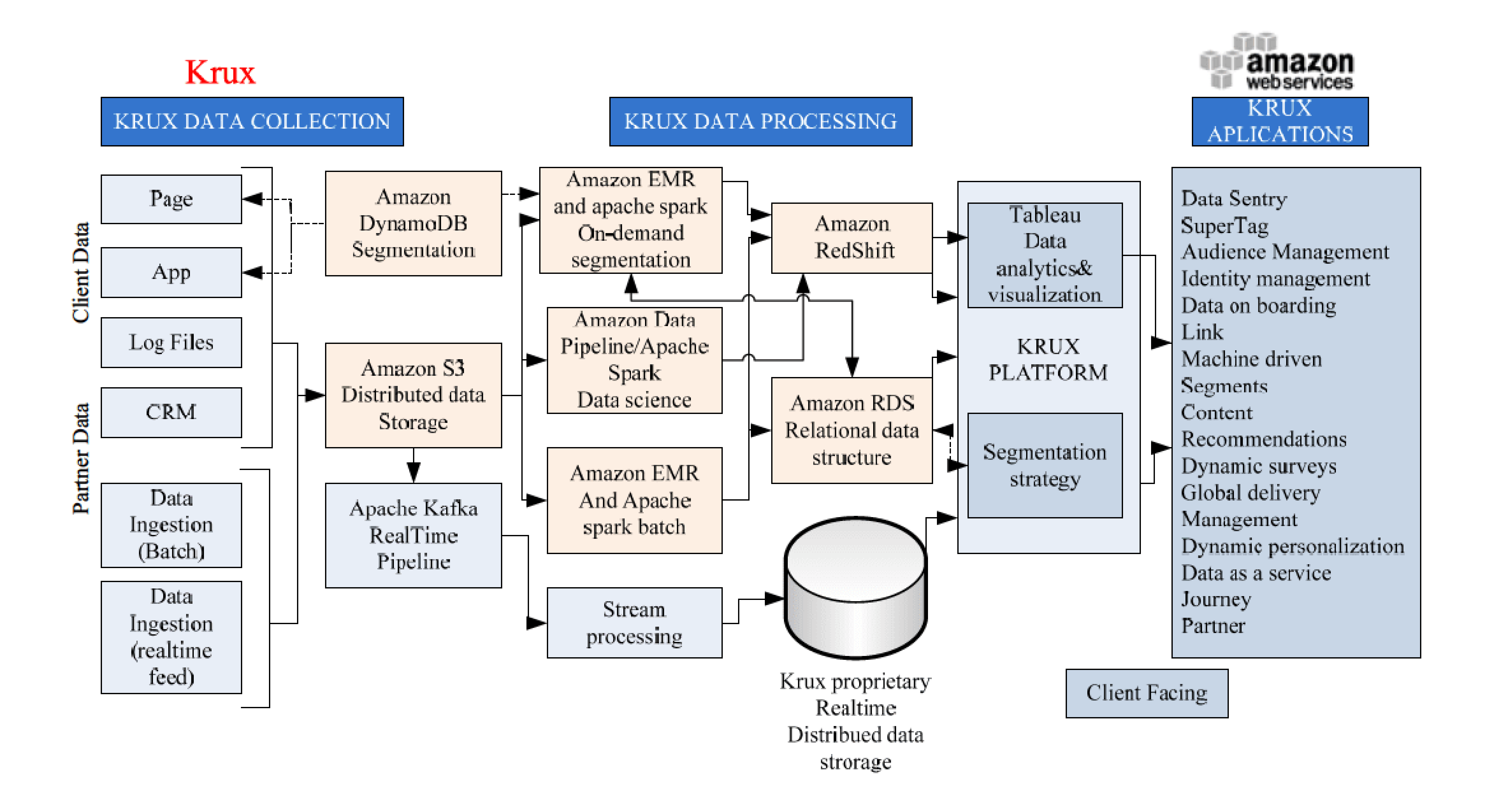


图 7-2　krux 大数据解决方案

7.3 Spark 在商业网站中的应用

为了满足挖掘分析与交互式实时查询的计算需求，腾讯大数据使用了 Spark 平台来支持挖掘分析类计算、交互式实时查询计算以及允许误差范围的快速查询计算。目前腾讯大数据拥有超过 200 台的 Spark 集群，并独立维护 Spark 和 Shark 分支。Spark 集群已稳定运行 2 年，积累了大量的案例和运营经验，另外多个业务的大数据查询与分析应用，已在陆续上线并稳定运行。在 SQL 查询性能方面，普遍比 MapReduce 高出 2 倍以上，利用内存计算和内存表的特性，性能至少在 10 倍以上。在迭代计算与挖掘分析方面，精准推荐将小时和天级别的模型训练转变为 Spark 的分钟级别的训练，同时简洁的编程接口使得算法实现比 MR 在时间成本和代码量上高出许多[62]。

1. 腾讯

广点通是最早使用 Spark 的应用之一。腾讯大数据精准推荐借助 Spark 快速迭代的优势，围绕“数据+算法+系统”这套技术方案，实现了在“数据实时采集、算法实时训练、系统实时预测”的全流程实时并行高维算法，最终成功应用于广点通 pCTR 投放系统上，支持每天上百亿的请求量。

基于日志数据的快速查询系统业务构建于 Spark 之上的 Shark，利用其快速查询以及内存表等优势，承担了日志数据的即席查询工作。在性能方面，普遍比 Hive 高 2~10 倍，如果使用内存表的功能，性能将会比 Hive 快百倍。

2. Yahoo

Yahoo 将 Spark 用在 Audience Expansion 中。Audience Expansion 是广告中寻找目标用户的一种方法：首先广告者提供一些观看了广告并且购买产品的样本客户，据此进行学习，寻找更多可能转化的用户，对他们定向广告。Yahoo 采用的算法是 logistic regression。同时，由于有些 SQL 负载需要更高的服务质量，又加入了专门跑 Shark 的大内存集群，用于取代商业 BI/OLAP 工具，承担报表/仪表盘和交互式/即席查询，同时与桌面 BI 工具对接。目前在 Yahoo 部署的 Spark 集群有 112 台节点，9.2TB 内存。

3. 淘宝

阿里搜索和广告业务，最初使用 Mahout 或者自己写的 MR 来解决复杂的机器学习，导致效率低下而且代码不易维护。淘宝技术团队使用了 Spark 来解决多次迭代的机器学习算法、高计算复杂度的算法等。将 Spark 运用于淘宝的推荐相关算法上，同时还利用 Graphx 解决了许多生产问题，包括以下计算场景：基于度分布的中枢节点发现、基于最大连通图的社区发现、基于三角形计数的关系衡量、基于随机游走的用户属性传播等。

4. 优酷土豆

优酷土豆在使用 Hadoop 集群的突出问题主要包括：第一是商业智能 BI 方面，分析师提交任务之后需要等待很久才得到结果；第二就是大数据量计算，比如进行一些模拟广告投放，计算量非常大的同时对效率要求也比较高，最后就是机器学习和图计算的迭代运算也是需要耗费大量资源且速度很慢。

最终发现这些应用场景并不适合在 MapReduce 里面去处理。通过对比，发现 Spark 性能比 MapReduce 提升很多。首先，交互查询响应快，性能比 Hadoop 提高若干倍；模拟广告投放计算效率高、延迟小（同 Hadoop 比延迟至少降低一个数量级）；机器学习、图计算等迭代计算，大大减少了网络传输、数据落地等，极大地提高了计算性能。目前 Spark 已经广泛使用在优酷土豆的视频推荐（图计算）、广告业务等方面。

7.4 Spark 在 Yahoo!的应用

Yahoo!在 Spark 技术的研究与应用方面始终处于领先地位，它将 Spark 应用于公司的各种产品之中。移动 App、网站、广告服务、图片服务等服务的后端实时处理框架均采用了 Spark+Shark 的架构[105]。Yahoo!使用 Spark 进行数据分析的整体架构如图 7-3 所示。

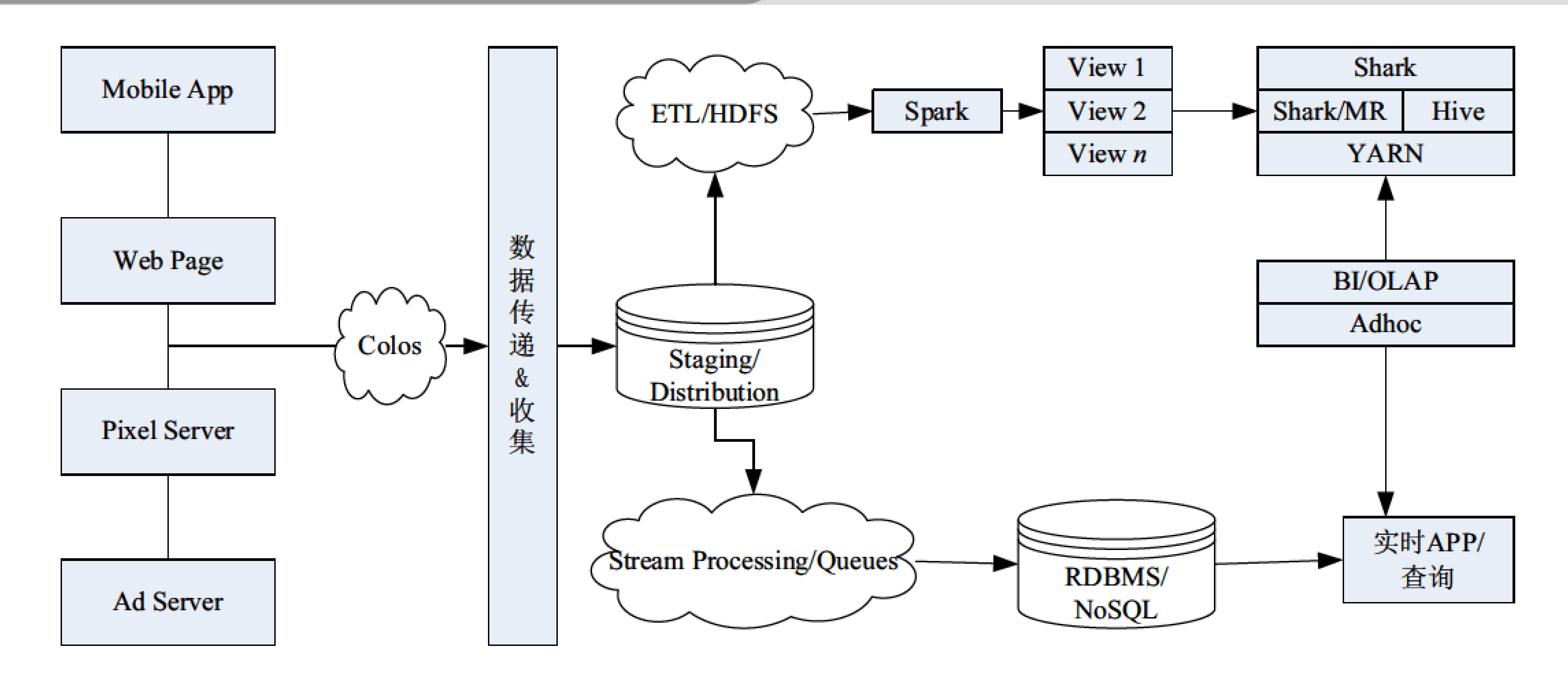


图 7-3 Yahoo!使用 Spark 进行数据分析的整体架构

为了让 Hadoop 和 Spark 的任务共存，整个数据分析栈构建在 YARN 之上。主要包含两个主要模块：

（1）使用 MapReduce 和 Spark+Shark 混合架构。由于 MapReduce 适合进行 ETL 处理，还保留 Hadoop 进行数据清洗和转换。数据在 ETL 之后加载进 HDFS/HCat/Hive 数据仓库存储，之后可以通过 Spark、Shark 进行 OLAP 数据分析。

（2）使用 Spark Streaming+Spark+Shark 架构进行处理。实时流数据源源不断经过 Spark Steaming 初步处理和分析之后，将数据追加进关系数据库或者 NoSQL 数据库。之后，结合历史数据，使用 Spark 进行实时数据分析。

7.5 Spark 在 Amazon EC2 上运行

Spark-ec2 可以管理多个命名集群。你可以用它来启动一个新集群（需要提供集群大小和集群名称），关闭一个已有的集群，或者登录到一个集群。每一个集群的机器将会被划分到不同的 EC2 安全组（EC2 security groups）当中，而这些安全组的名字是由集群的名称派生而来。例如，对于一个命名为 test 的集群，其主节点（Master）将被分到一个叫 test-master 的安全组，而其他从节点（Slave）将被分配到 test-slaves 安全组。spark-ec2 脚本会自动根据你提供的集群名称来创建安全组。你可以在 EC2 的控制台（Amazon EC2 Console）中使用这些名字。

Spark 的 ec2 目录下有一个 spark-ec2 脚本，可以帮助你在 Amazon EC2 上启动、管理、关闭 Spark 集群。该脚本能在 EC2 集群上自动设置好 Spark 和 HDFS。本文将会详细描述如何利用 spark-ec2 脚本来启动和关闭集群，以及如何在集群提交作业。当然，首先必须在 Amazon Web Services site 上注册一个 EC2 的账户。

1. 准备工作

首先，你需要创建 Amazon EC2 key pair 。这需要登录 Amazon Web Services 账号，在 AWS 控制台（AWS console）上单击侧边栏上的 Key Pairs 来创建，并下载。同时，你要确保给这私匙文件附上 600 权限（即：可读可写）以便使用 ssh 登录。

使用 spark-ec2 的时候，一定要设置好这两个环境变量，AWS_ACCESS_KEY_ID 和 AWS_SECRET_ACCESS_KEY，并使其指向你的 Amazon EC2 access key ID 和 secret access key。这些都可以在 AWS 主页（AWS homepage）上单击 Account→Security Credentials→Access Credentials 获得。

2. 启动集群

切换到你下载的 spark 的 ec2 目录下。

运行命令./spark-ec2 -k <keypair> -i <key-file> -s <num-slaves> launch <cluster-name>，其中<keypair>是你的 Amazon EC2 key pair 的名字（你创建 Amazon EC2 key pair 的时候所指定的名字），<key-file>是 Amazon EC2 key pair 的私钥（private key）文件，<num-slaves>是 slave 节点个数（至少是 1），<cluster-name>是你指定的集群名称。

例如：

```
bash export AWS_SECRET_ACCESS_KEY=AaBbCcDdEeFGgHhIiJjKkLlMmNnOoPpQqRrSsTtU \
export AWS_ACCESS_KEY_ID=ABCDEFG1234567890123

./spark-ec2 --key-pair=awskey \
--identity-file=awskey.pem \
--region=us-west-1 \
--zone=us-west-1a \
launch my-spark-cluster
```

集群启动完成后，检查一下集群调度器是否启动，同时，你可以在 Web UI 上查看是否所有的 Slave 节点都正确地展示出来了，Web UI 的链接在脚本执行完以后会打印在屏幕上（通常这个链接是 http://<master-hostname>:8080）。

你可以运行./spark-ec2 –help 来查看更多的选项。以下是比较重要的一些选项：

- –instance-type=<instance-type> ：可以指定 EC2 机器的实例类型。目前，该脚本只支持 64-bit 的实例类型。
- –region=<ec2-region>：可以指定 EC2 集群部署于哪个地域，默认地域是 us-east-1。
- –zone=<ec2-zone>：可以指定 EC2 集群实例部署在哪些地区（EC2 的可用地区）。指定这个参数时注意，有时候因为在某些地区可能出现容量不够，因此你可能需要在其他地区启动 EC2 集群。
- –ebs-vol-size=<GB>：可以在每个节点上附加一个 EBS（弹性可持续存储）卷，并指定其总容量，这些存储时可持久化的，即使集群重启也不会丢失。
- –spot-price=<price> ：将启动竞价型实例（Spot Instances）工作节点，这些节点可以

按需分配，可竞价，并且可以设定竞价最高价格（以美元计）。

- –spark-version=<version> ：可以在集群中预先加载指定版本的 spark。<version>可以是一个版本号（如：0.7.3）或者是一个 git hash 值。默认会使用最新版本的 spark。
- –spark-git-repo=<repository url> ：可以指定一个自定义的 git 库，从而下载并部署该 git 库中特定的 spark 构建版本，默认使用 Apache Github mirror 。如果同时指定了 spark 版本，那么–spark-version 参数值不能使用版本号，而必须是一个 git 提交对应的 git commit hash（如：317e114）。

如果启动过程中由于某些原因失败了（如：没有给 private key 文件设定正确的文件权限），你可以用–resume 选项来重启并继续已有集群的部署过程。

3. 在 VPC（Amazon Virtual Private Cloud）上启动集群

运行 ./spark-ec2 -k <keypair> -i <key-file> -s <num-slaves> –vpc-id=<vpc-id> -subnet-id=<subnet-id> launch <cluster-name>，其中，<keypair>是你的 EC2 key pair（之前已经创建的），<key-file>是 key pair 中的私钥文件，<num-slaves> 是从节点个数（如果你是第一次用，可以先设成 1），<vpc-id> 是 VPC 的名称，<subnet-id> 是你的子网名称，最后<cluster-name>是你的集群名称。

例如：

```
bash export AWS_SECRET_ACCESS_KEY=AaBbCcDdEeFGgHhIiJjKkLlMmNnOoPpQqRrSsTtU \
export AWS_ACCESS_KEY_ID=ABCDEFG1234567890123

./spark-ec2 --key-pair=awskey \
--identity-file=awskey.pem \
--region=us-west-1 \
--zone=us-west-1a \
--vpc-id=vpc-a28d24c7 \
--subnet-id=subnet-4eb27b39 \
--spark-version=1.1.0 \
launch my-spark-cluster
```

4. 运行应用

转到你下载的 spark 的 ec2 目录下。

执行 ./spark-ec2 -k <keypair> -i <key-file> login <cluster-name>，远程登录到你的 EC2 集群，其中，<keypair> 和<key-file> 的说明见本文上面（这里只是为了方便说明，你也可以使用 EC2 的控制台）。

如果需要把代码或数据部署到 EC2 集群中，你可以在登录后，使用脚本 ~/spark-ec2/copy-dir，并指定一个需要 RSYNC 同步到所有从节点（slave）上的目录。

如果你的应用需要访问一个很大的数据集，最快的方式就是从 Amazon S3 或者 Amazon

EBS 设备上加载这些数据，然后放到你集群中的 HDFS 上。spark-ec2 脚本已经为你设置好了一个 HDFS，其安装目录为/root/ephemeral-hdfs，并且可以使用该目录下的 bin/hadoop 脚本访问。需要特别注意的是，这个 HDFS 上的数据，在集群停止或重启后，会被自动删掉。

集群中也有可以持久的 HDFS，其安装路径为/root/persistent-hdfs，这个 HDFS 保存的数据即使集群重启也不会丢失。但一般情况下，这个 HDFS 在每个节点上可使用的空间较少（约为 3GB），你可以用 spark-ec2 的选项–ebs-vol-size 来指定每个节点上持久化 HDFS 所使用的空间大小。

最后，如果你的应用出错，你可以看看该应用在 slave 节点的日志，日志位于调度器工作目录下（/root/spark/work）。当然，你也可以通过 Web UI（http://<master-hostname>:8080）查看一下集群状态。

5. 配置

你可以编辑每个节点上的/root/spark/conf/spark-env.sh 文件来设置 Spark 配置选项（如：JVM 选项参数），这个文件一旦更改，你必须将其复制到集群中所有节点上。最简单的方式仍然是使用 copy-dir 脚本。首先，编辑主节点（Master）上的 spark-env.sh 文件，然后，运行~/spark-ec2/copy-dir/root/spark/conf，将 conf 目录 RSYNC 到所有工作节点上。

6. 终止集群

如果 EC2 节点被关闭后，是没有办法恢复其数据的！所以，请务必确保在关闭节点之前，将所有重要的数据复制出来，备份好。

切换到 spark 下的 ec2 目录。

运行命令 ./spark-ec2 destroy <cluster-name>。

7. 暂停和重启集群

spark-ec2 脚本同样支持暂停集群。这种情况下，集群实例所使用的虚拟机都是被停止，但不会销毁，所以虚拟机上临时盘数据都会丢失，但 root 分区以及持久 HDFS（persistent-hdfs）上的数据不会丢失。停止机器实例不会多花 EC2 周期（意味着不用为机器实例付费），但会持续 EBS 存储的计费。

要停止一个集群，你需要切到 ec2 目录下，运行 ./spark-ec2 –region=<ec2-region> stop <cluster-name>。

如果过后又要重启，请运行 ./spark-ec2 -i <key-file> –region=<ec2-region> start <cluster-name>。

如果需要最终销毁这个集群，并且不再占用 EBS 存储空间，需要运行 ./spark-ec2 –region=<ec2-region> destroy <cluster-name>。

8. 限制

对“集群计算”的支持有个限制，即无法指定一个局部群组。不过，你可以在<cluster-name>-slaves 群组中手工启动一些 Slave 节点，然后用 spark-ec2 launch –resume 这个命令将手工启动的节点组成一个集群。

9. 访问 S3 上的数据

Spark 文件接口允许你通过相同的 URI 格式访问所有在 Amazon S3 上的数据，当然这些数据格式必须是 Hadoop 所支持的。可以通过这种 URI 格式指定 S3 路径 s3n://<bucket>/path。在启动 Spark 集群的时候，可以使用选项–copy-aws-credentials 来指定访问 S3 的 AWS 证书。更完整的访问 S3 所需的 Hadoop 库可以在这里查看 Hadoop S3 page。

另外，访问 S3 的时候，不仅可以将单个文件路径作为输入，同时也可以将整个目录路径作为输入。

7.6 淘宝应用 Spark on YARN 架构

对于基于 YARN 的 Spark 作业，首先由客户端生成作业信息，提交给 ResourceManager，ResourceManager 在某一 NodeManager 汇报时把 AppMaster 分配给 NodeManager，NodeManager 启动 SparkAppMaster，SparkAppMaster 启动后初始化作业，然后向 ResourceManager 申请资源，申请到相应资源后，SparkAppMaster 通过 RPC 让 NodeManager 启动相应的 SparkExecutor，SparkExecutor 向 SparkAppMaster 汇报并完成相应的任务。此外，SparkClient 会通过 AppMaster 获取作业运行状态。目前，淘宝数据挖掘与计算团队通过 Spark on YARN 已实现 MLR、PageRank 和 JMeans 算法，其中 MLR 已作为生产作业运行[106]。Spark on YARN 架构如图 7-4 所示。

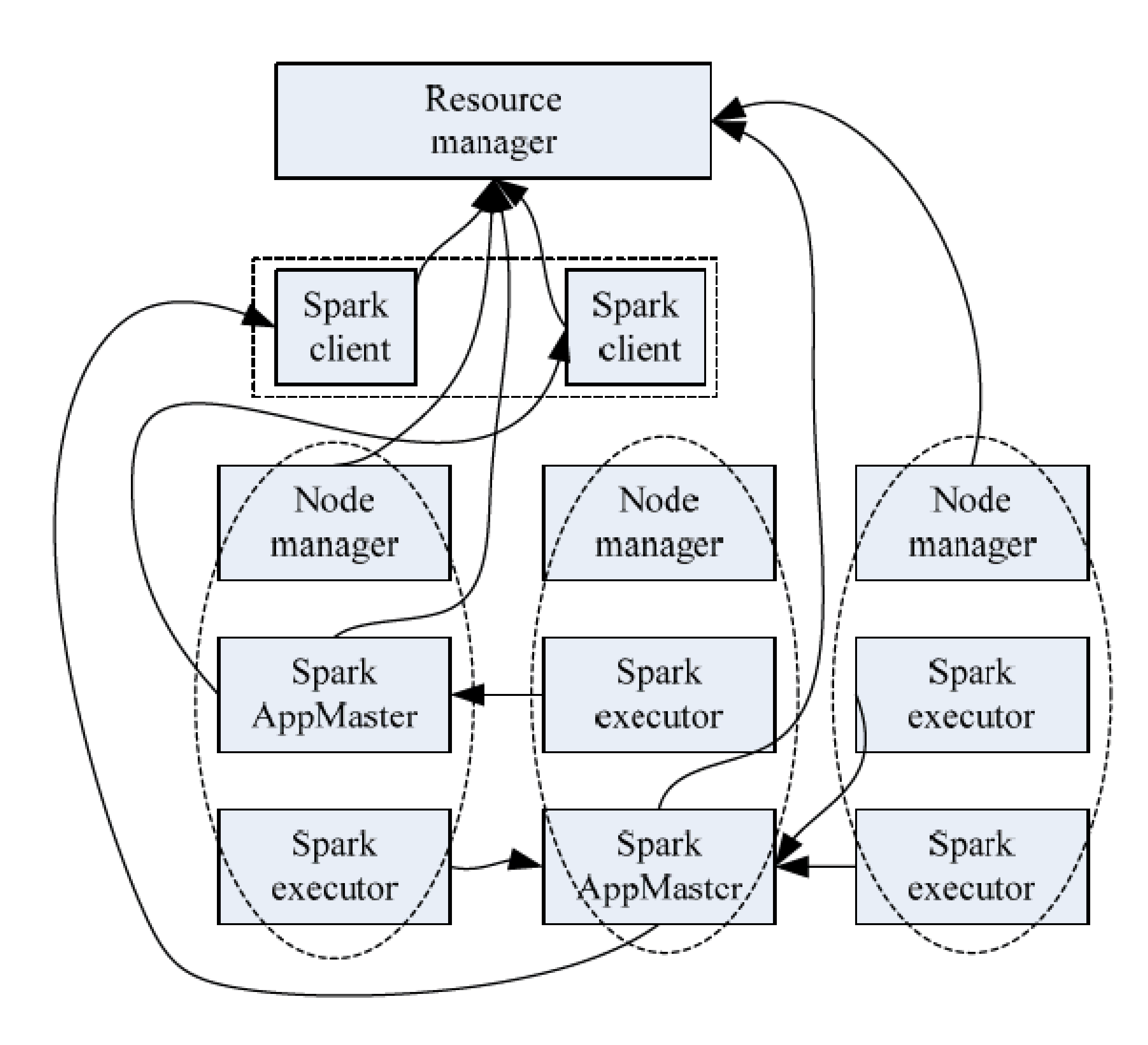


图 7-4 Spark on YARN 架构

7.7 腾讯云大数据解决方案

在互联网+背景下，各级政府正在探索如何将政务信息化过程中积累的海量数据发挥出更大价值，为民众提供更便捷、更智能的服务，优化提升行政效率和质量。然而，传统技术和社区方案难以满足多种海量数据处理能力和高可靠、高安全需求，难以发挥内部外部数据整合优势。

腾讯大数据政务方案，可以快速地为政府机构提供一站式大数据方案，快速接入内部和外部数据，从数据处理、分析到展示，充分挖掘政务数据潜力，提供交互式展示工具，助力外网门户服务或内部政务应用。还可结合腾讯海量信息数据，提供人群画像、区域人流分析等公共服务，助力政府高效行政，快速决策。腾讯云政务大数据处理架构（腾讯网）如图 7-5 所示[107]。

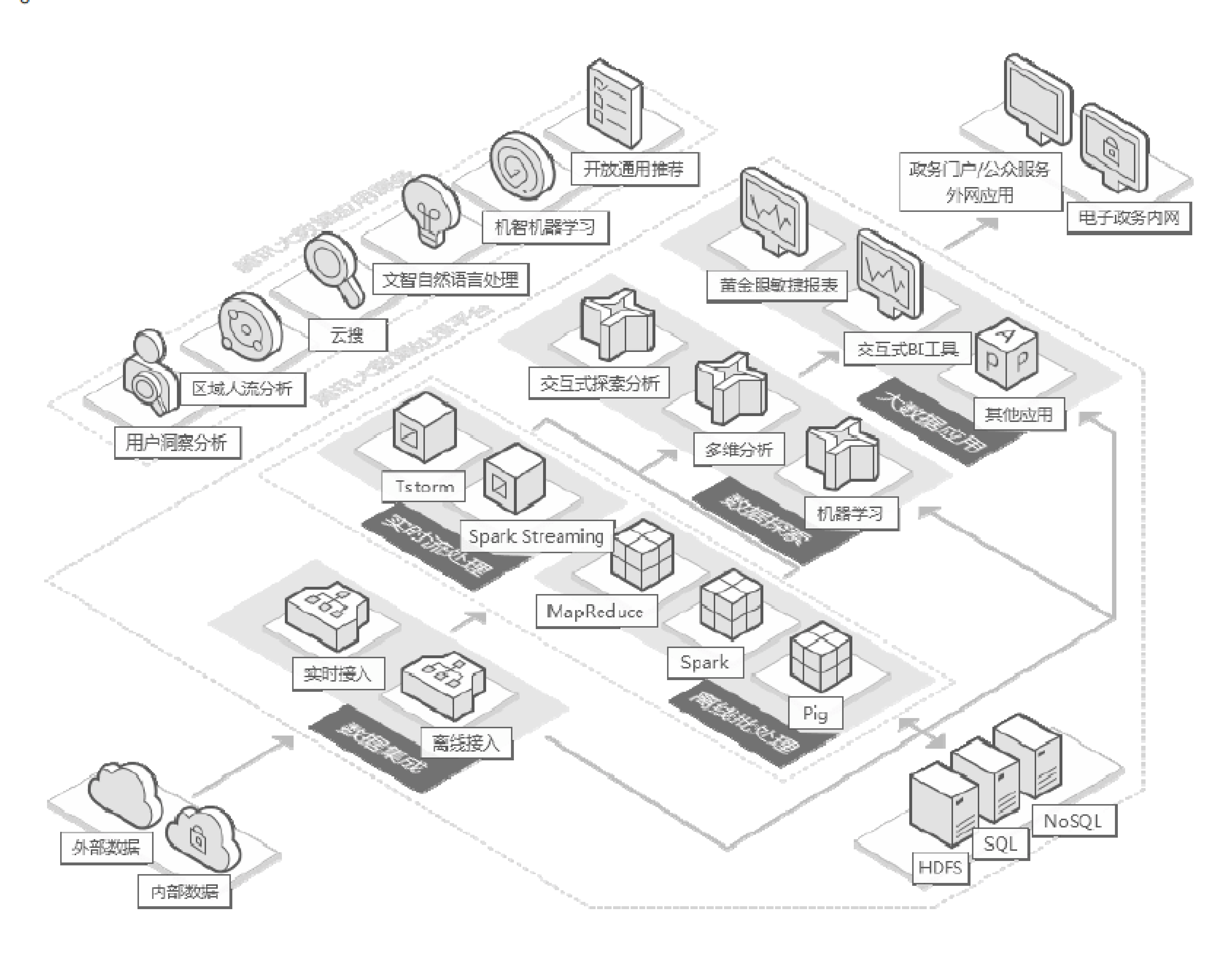


图 7-5　腾讯云政务大数据处理架构

一站式搭建政务大数据处理平台、快速接入多个政务业务系统、实时或离线数据分析、即刻洞察展示政务数据，使得政府客户能够将精力集中在分析业务数据本身，而非平台的搭建和运维。完备的合规性和安全分级管控全面保护数据安全，快速提交交互式报表，辅助政务决策。

离线批处理计算中，支持 MapReduce、Hive、Pig 等批处理计算作业，采用 Spark 分布式内存计算框架，以支持复杂的数据挖掘算法和图计算算法，Storm 流式任务作业引擎，覆盖实时要求极高的流式作业场景。支持基于 Spark 上的 Spark Streaming，满足毫秒级的实时计算场景需求，如实时推荐、用户行为分析等。

7.8 雅虎开源 TensorFlowOnSpark

2016 年雅虎结合了大数据和机器学习领域的两大明星，将内存数据处理框架 Spark 与深度学习框架 Caffe 集成。在 Spark 中编写的应用程序将使用 Caffe 的训练功能，或者使用经过训练的模型来进行 Spark 本地机器学习无法实现的预测[63]。

2017 年，雅虎又发了一波大招，最新的 Yahoo 开源项目 TensorFlowOnSpark（简称 TFoS， Github 地址：https://github.com/yahoo/TensorFlowOnSpark），再次融合了深度学习和大数据框架，能够更有效地大规模运行，并且几乎没有改变现有的 Spark 应用程序。

TFoS 被设计为在现有的 Spark 和 Hadoop 集群上运行，并使用现有的 Spark 库，如 SparkSQL 或 Spark 的 MLlib 机器学习库。雅虎声称现有的 TensorFlow 程序不需要大量修改就可以使用 TFoS。通常，这种改变少于 10 行 Python 代码，TensorFlow 的并行实例可以直接相互通信，而无须通过 Spark 本身。数据可以从 TensorFlow 的本地设备中获取，以便从 HDFS 或通过 Spark 读取。

我们举例说明如何将 tensorflowonspark 应用于 Sparkstandalone 簇。

1. 克隆 TensorFlowOnOnSpark 代码

```
git clone --recurse-submodules https://github.com/yahoo/TensorFlowOnSpark.git
cd TensorFlowOnSpark
git submodule init
git submodule update --force
git submodule foreach --recursive git clean -dfx

cd TensorFlowOnSpark
export TFoS_HOME=$(pwd)
pushd src
zip -r ../tfspark.zip *
popd
```

2. 安装 Spark

按照网址 http://spark.apache.org/downloads.html 安装 Apache Spark 2.1.0。

```
${TFoS_HOME}/scripts/local-setup-spark.sh
rm spark-2.1.0-bin-hadoop2.7.tar
export SPARK_HOME=$(pwd)/spark-2.1.0-bin-hadoop2.7
export PATH=${SPARK_HOME}/bin:${PATH}
```

3. 安装 TensorFlow

在 Mac OS 安装 TensorFlow 示例：

```
export
    TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow
    -0.12.1-py2-none-any.whl
sudo pip install --upgrade $TF_BINARY_URL
```

Test TensorFlow：

```
# download MNIST files, if not already done
mkdir ${TFoS_HOME}/mnist
pushd ${TFoS_HOME}/mnist
curl -O "http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz"
curl -O "http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz"
curl -O "http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz"
curl -O "http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz"
popd

python
    ${TFoS_HOME}/tensorflow/tensorflow/examples/tutorials/mnist/mnist_with_sum
    maries.py --data_dir ${TFoS_HOME}/mnist
```

4. 启动 standalone Spark 集群

启动 Master：

```
${SPARK_HOME}/sbin/start-master.sh
Start one or more workers and connect them to the master via master-spark-URL.
    Go to MasterWebUI, make sure that you have the exact number of workers
    launched.
export MASTER=spark://$(hostname):7077
export SPARK_WORKER_INSTANCES=2
export CORES_PER_WORKER=1
export TOTAL_CORES=$((${CORES_PER_WORKER}*${SPARK_WORKER_INSTANCES}))
${SPARK_HOME}/sbin/start-slave.sh -c $CORES_PER_WORKER -m 3G ${MASTER}
```

5. MNIST zip 文件的转换

```
cd ${TFoS_HOME}
rm -rf examples/mnist/csv
${SPARK_HOME}/bin/spark-submit \
--master ${MASTER} \
${TFoS_HOME}/examples/mnist/mnist_data_setup.py \
--output examples/mnist/csv \
--format csv
```

```
ls -lR examples/mnist/csv
```

6. 运行分布式 MNIST 训练（使用 feed_dict）

```
# rm -rf mnist_model
${SPARK_HOME}/bin/spark-submit \
--master ${MASTER} \
--py-files
    ${TFoS_HOME}/tfspark.zip,${TFoS_HOME}/examples/mnist/spark/mnist_dist.py \
--conf spark.cores.max=${TOTAL_CORES} \
--conf spark.task.cpus=${CORES_PER_WORKER} \
--conf spark.executorEnv.JAVA_HOME="$JAVA_HOME" \
${TFoS_HOME}/examples/mnist/spark/mnist_spark.py \
--cluster_size ${SPARK_WORKER_INSTANCES} \
--images examples/mnist/csv/train/images \
--labels examples/mnist/csv/train/labels \
--format csv \
--mode train \
--model mnist_model

ls -l mnist_model
```

7. 运行分布式 MNIST 推理（使用 feed_dict）

```
# rm -rf predictions
${SPARK_HOME}/bin/spark-submit \
--master ${MASTER} \
--py-files
    ${TFoS_HOME}/tfspark.zip,${TFoS_HOME}/examples/mnist/spark/mnist_dist.py \
--conf spark.cores.max=${TOTAL_CORES} \
--conf spark.task.cpus=${CORES_PER_WORKER} \
--conf spark.executorEnv.JAVA_HOME="$JAVA_HOME" \
${TFoS_HOME}/examples/mnist/spark/mnist_spark.py \
--cluster_size ${SPARK_WORKER_INSTANCES} \
--images examples/mnist/csv/test/images \
--labels examples/mnist/csv/test/labels \
--mode inference \
--format csv \
--model mnist_model \
--output predictions
```

```
less predictions/part-00000
The prediction result should look like:
2017-02-10T23:29:17.009563 Label: 7, Prediction: 7
2017-02-10T23:29:17.009677 Label: 2, Prediction: 2
2017-02-10T23:29:17.009721 Label: 1, Prediction: 1
2017-02-10T23:29:17.009761 Label: 0, Prediction: 0
2017-02-10T23:29:17.009799 Label: 4, Prediction: 4
2017-02-10T23:29:17.009838 Label: 1, Prediction: 1
2017-02-10T23:29:17.009876 Label: 4, Prediction: 4
2017-02-10T23:29:17.009914 Label: 9, Prediction: 9
2017-02-10T23:29:17.009951 Label: 5, Prediction: 6
2017-02-10T23:29:17.009989 Label: 9, Prediction: 9
2017-02-10T23:29:17.010026 Label: 0, Prediction: 0
```

8. 与 Jupyter Notebook 的互动学习

通过 Jupyter Notebooks 安装附加软件：

```
sudo pip install jupyter jupyter[notebook]
```

在 Master node 节点启动 IPython notebook：

```
pushd ${TFoS_HOME}/examples/mnist
PYSPARK_DRIVER_PYTHON="jupyter" \
PYSPARK_DRIVER_PYTHON_OPTS="notebook --no-browser --ip=`hostname`" \
pyspark  --master ${MASTER} \
--conf spark.cores.max=${TOTAL_CORES} \
--conf spark.task.cpus=${CORES_PER_WORKER} \
--py-files
    ${TFoS_HOME}/tfspark.zip,${TFoS_HOME}/examples/mnist/spark/mnist_dist.py \
--conf spark.executorEnv.JAVA_HOME="$JAVA_HOME"
```

9. 关闭 Spark 集群

```
${SPARK_HOME}/sbin/stop-slave.sh
${SPARK_HOME}/sbin/stop-master.sh
```

7.9 阿里云 E-MapReduce

E-MapReduce 是构建于阿里云 ECS 弹性虚拟机之上，利用开源大数据生态系统，包括 Hadoop、Spark、HBase，为用户提供集群、作业、数据等管理的一站式大数据处理分析服务[108]。

E-MapReduce 具有如下特性。

- 控制台新增交互式工作台：直接在 Web 上编写代码，立即运行，并查看结果。
- 支持开源软件界面直接查看：通过控制台直接查看 YARN、Ganglia 的界面。
- 新增独享实例支持：为你的集群提供稳定的带宽和 CPU，只在华东和华北的部分可用区。
- 执行计划支持重跑：可以指定执行计划从任意节点开始重跑，方便作业失败后的手工恢复。
- 增加 MetaService 支持：在访问外部云服务的时候，不再需要显式地输入 AK，操作更加方便。
- 增加集群脚本：随时随地可以通过集群脚本来对集群的所有节点进行变更运行环境、安装新的组件等。
- 增加 Tez 支持：2.1.0 及以上版本 Hive 的执行时，可以选择 Tez 作为执行引擎，极大地提高 MR 作业的效率。
- 增加 Sqoop、SparkSQL、Shell 作业：使用 Sqoop 来同步数据、SparkSQL 来执行高效的 SQL 作业、Shell 来执行任意脚本命令。

1. E-MapReduce 功能

（1）自动化按需创建集群

- 自由选择机器配置（CPU、内存）、磁盘类型和容量。
- 自由选择服务器规模，包括 Master 和 Core 的数量。
- 根据业务量的上升可对集群动态扩容。
- 自由选择开源大数据生态软件组合和版本，目前包括 Hadoop 和 Spark。
- 自由选择启动集群的方式，分为临时集群和长时间运行集群。

（2）支持丰富的作业类型

- MapReduce：离线处理作业。
- Hive：关系型分析查询作业。
- Pig：数据清洗、ETL 等脚本作业。
- Spark MLlib：基于 Spark 的机器学习作业。
- Spark GraphX：基于 Spark 的图处理作业。
- Spark Streaming：基于 Spark 的在线/流式作业。
- Spark SQL & DataFrames：基于 Spark 的数据科学交互式作业。

（3）灵活的作业执行计划

- 将作业（包括 Hadoop/Spark/Hive/Pig）任意组合成执行计划。
- 执行计划的执行策略有两种，分为立即执行和定时周期执行。

2. E-MapReduce 典型应用场景

以下图例为 E-MapReduce 典型应用场景。

（1）离线数据处理（如图 7-6 所示）

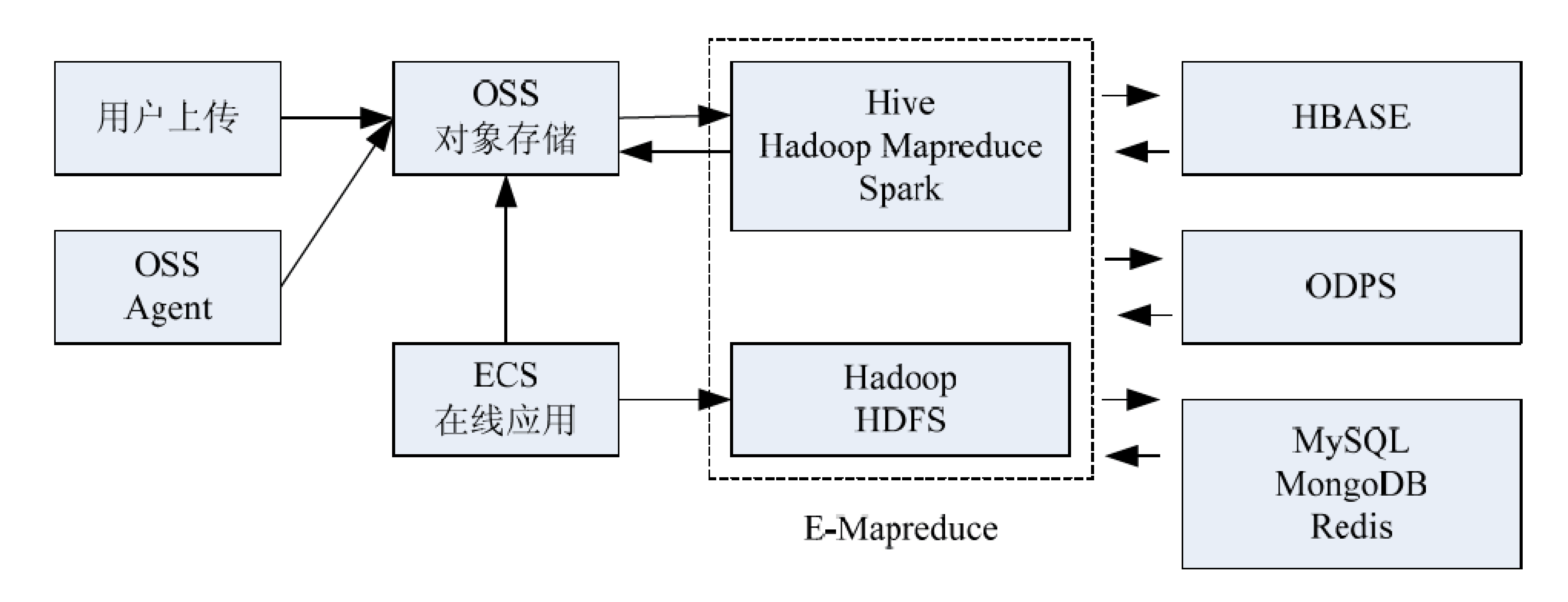


图 7-6 E-MapReduce 离线数据处理

（2）Ad hoc 数据分析（如图 7-7 所示）

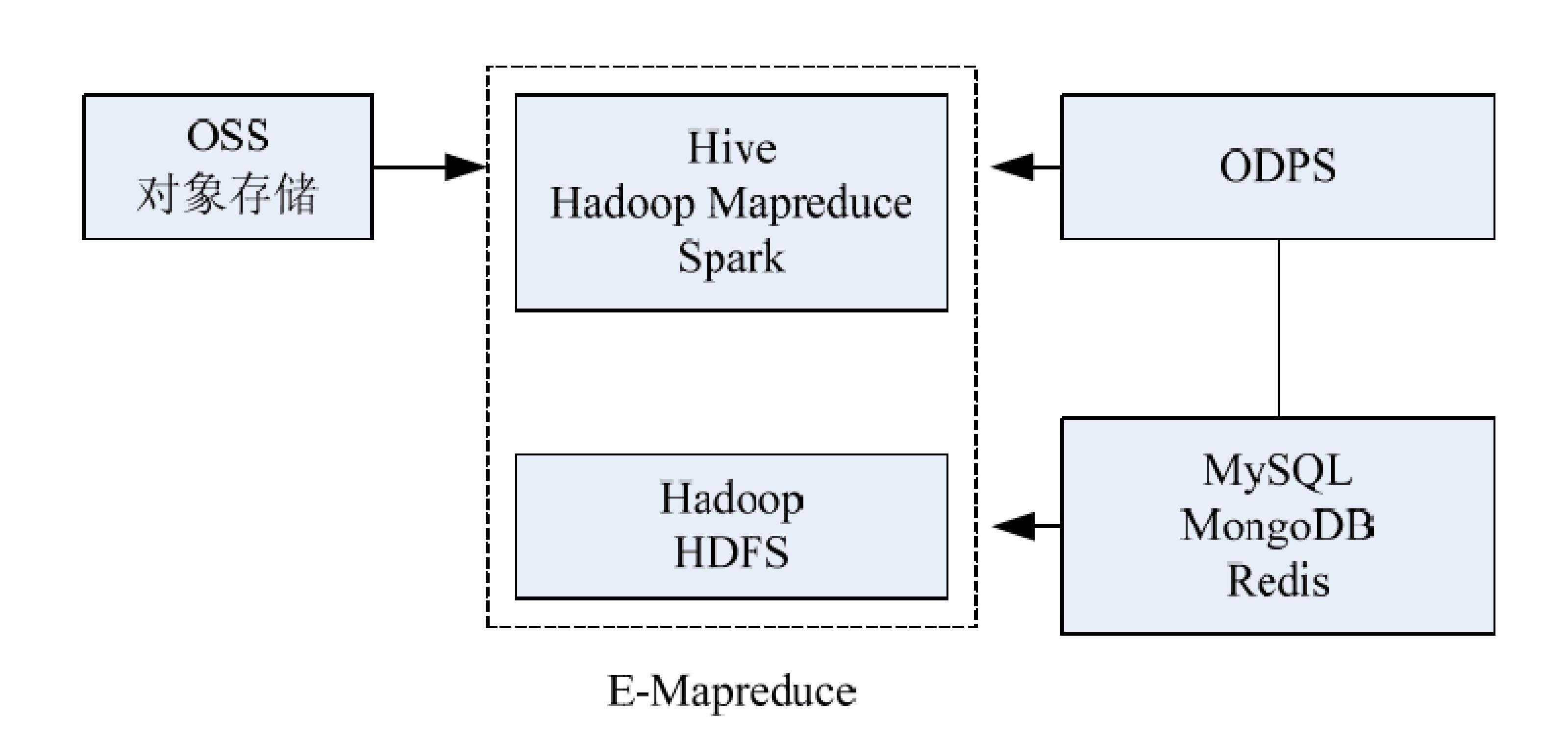


图 7-7 E-MapReduce Ad hoc 数据分析

（3）海量数据在线服务（如图 7-8 所示）

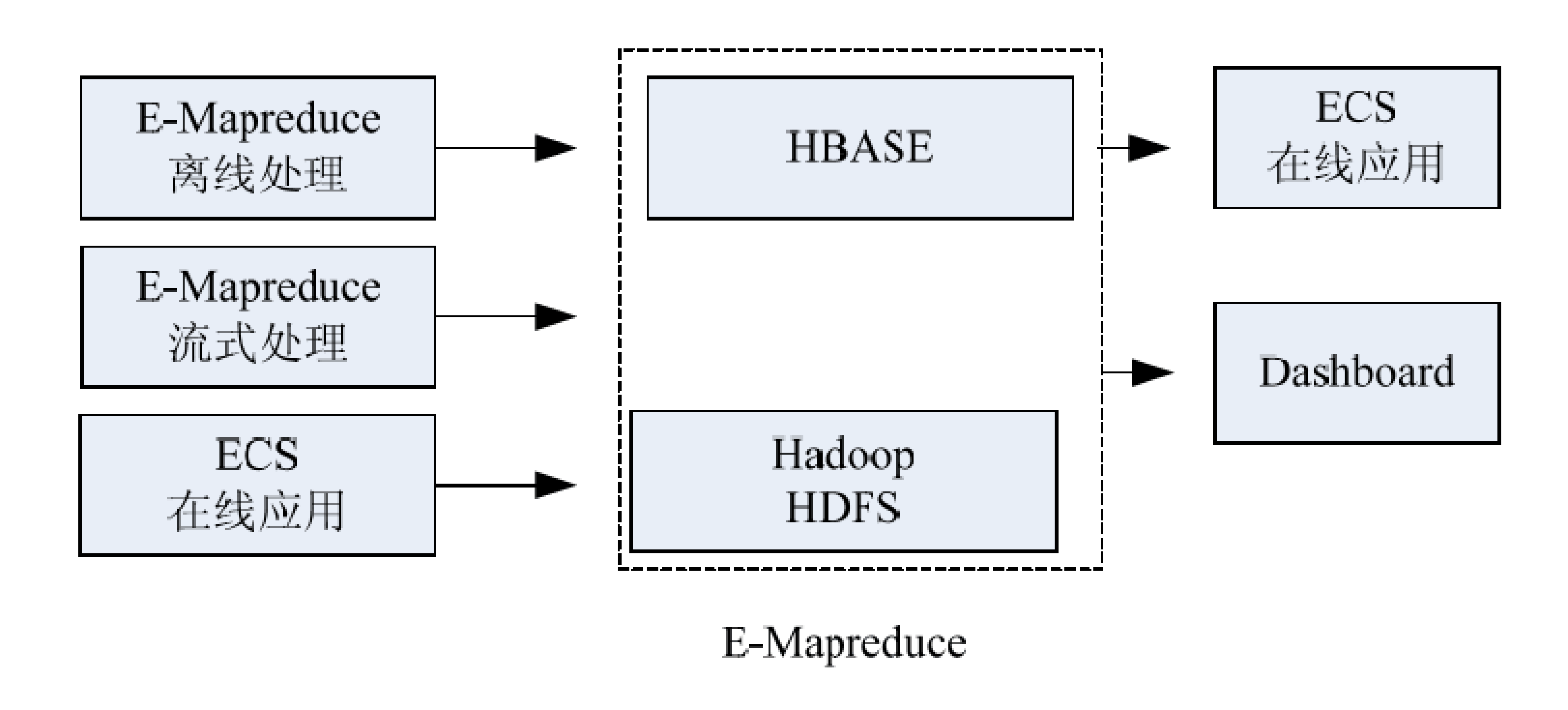


图 7-8 E-MapReduce 海量数据在线服务

（4）流式数据处理（如图 7-9 所示）

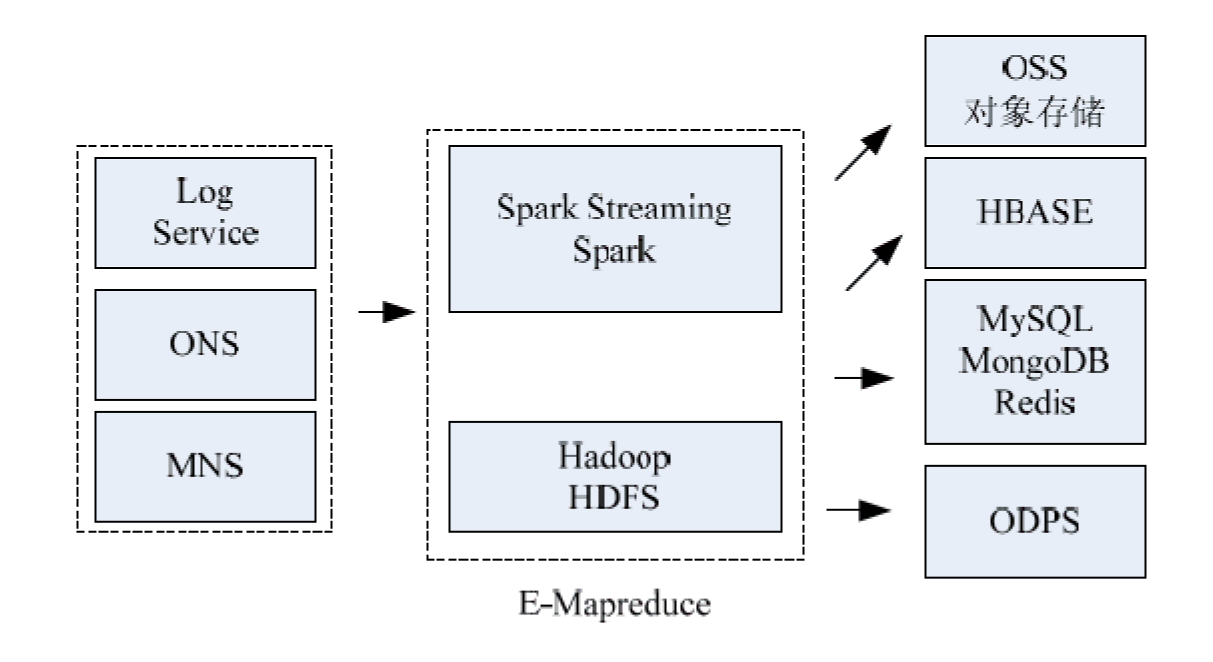


图 7-9　E-MapReduce 流式数据处理

7.10 SequoiaDB+Spark 打造一体化大数据平台

SequoiaDB 巨杉数据库作为在线数据存储系统，是国内第一个真正意义上整合分布式数据库与 Spark 的分布式计算存储框架。通过分布式的 SQL、索引、查询和计算模块整合，这一框架能够无缝适配当前主流大数据平台框架的发展趋势，也能够突出 SequoiaDB 的核心优势，为用户带来传统 Hadoop 体系无法提供的特性，提升企业数据的价值[2]。

1. SequoiaDB 是 Spark 底层数据源的首选

SequoiaDB 是一款文档型的分布式 NewSQL 数据库，其也是国内第一款完全自主研发、并且敢于开源的 NewSQL 数据库产品。SequoiaDB JSON 对象式的存储结构，带来灵活的数据结构；分布式的架构，使得存储容量可以动态调整；高可用和读写分离，则可以使得数据读写和离线数据分析分离，提升使用的效率；原生的 Spark-SequoiaDB Connector 连接器让 Spark 与 SequoiaDB 完美对接。

SequoiaDB 是一款 NewSQL 数据库，其可以在不同的物理节点之间对数据进行复制，并且允许用户指定使用哪一个数据备份。SequoiaDB 允许在同一集群同时运行数据分析和数据操作负载，并且保证最小的 I/O 和 CPU 使用率。

Spark-SequoiaDB Connector 是 Spark 的数据源，可以让用户使用 SparkSQL 对 SequoiaDB 的数据库集合中的数据进行读写。连接器用于 SequoiaDB 与 Spark 的集成，将无模式的存储模型、动态索引以及 Spark 集群的优势有机地结合起来。

2. SequoiaDB+Spark 打造一体化大数据平台

“Apache Spark 和 SequoiaDB 的联合解决方案，使得用户可以搭建一个在同一个物理集群中支持多种类型负载（如，SQL 语句和流处理）的统一平台。”

SequoiaDB+Spark 的一体化大数据平台，通过 SequoiaDB 与 Spark 架构的结合，实现了从数据的底层存储到数据的处理分析，最终实现数据展现的一体化平台。平台打通了数据从存储到最终展现的全过程，不仅大大降低了用户部署、使用的成本，简化了整个系统的操作和维护，同时更通过平台的一体化整合，大大减少了因为不同的产品、架构之间对接、通信等操作造成的系统效率和数据安全性降低。此外，Spark 的 SparkSQL 解析引擎，结合非结构化存储的 SequoiaDB，帮助现有的比较熟悉 SQL 语句的用户，能在基本不修改业务操作的情况下，顺利对接上 SequoiaDB+Spark 平台。

3. SequoiaDB+Spark 实战案例：产品精准推荐系统

这一系统使用分布式的 SequoiaDB，将所有用户的交易信息、操作信息进行了存储。这一存储的量级就已经达到了近 PB 级别。

之后，基于这些历史交易信息，平台就可以通过对这些数据的分析，对每个用户的交易行为进行预测，对用户进行分类和建模，最终根据分析的结果向每个用户推荐最适合的理财产品。

当用户模型系统通过分析所有的历史数据和日志，计算出需要推荐的产品时，这些用户特征也会作为这个用户的一个标签写入这个用户的信息中。这些新加入的用户标签，可以帮助前台的员工和产品推荐系统快速地分辨出每个顾客的兴趣和消费倾向。系统主架构图如图 7-10 所示。

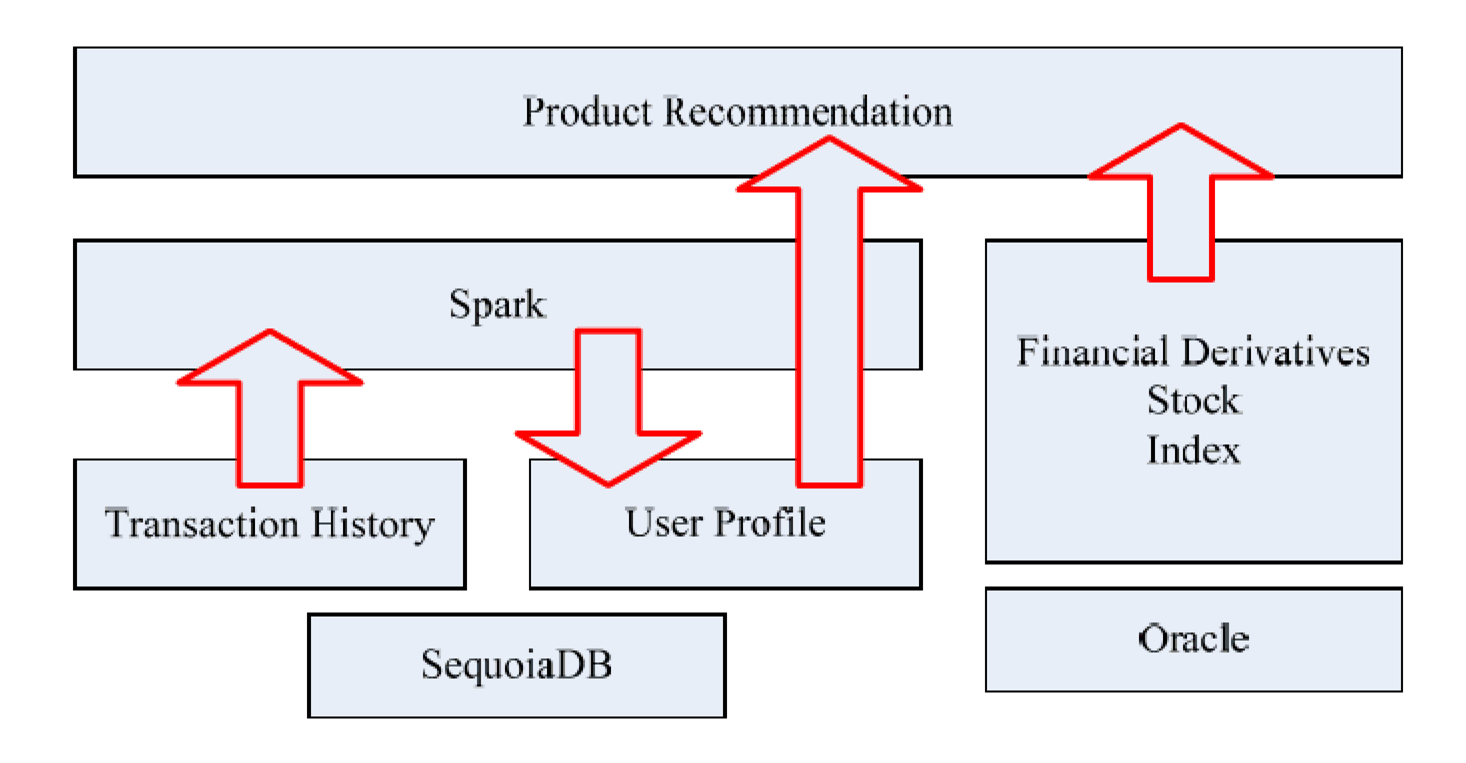


图 7-10　系统主架构图

7.11 本章小结

本章主要介绍 Spark 应用的典型案例，突出介绍 Spark 应用核心框架，提供了系统底层细节透明、编程接口简洁的分布式计算平台框架设计。根据 Spark 具有计算速度快、实时性高、容错性好等突出特点，基于 Spark 的应用已经逐步落地，尤其是在互联网领域，如淘宝、腾讯、网易等公司的发展已经成熟。同时电信、银行等传统行业也开始逐步试水 Spark 并取得了较好的效果。本章也对 Spark 的基本情况、架构、运行逻辑等进行了介绍。

第 8 章

大数据发展展望

大数据时代的出现，简单地说是海量数据同完美计算能力结合的结果。确切地说，是移动互联网、物联网产生了海量的数据，大数据计算技术完美地解决了海量数据的收集、存储、计算、分析的问题。大数据时代开启人类社会利用数据价值的另一个时代。

目前大数据在互联网公司主要应用在广告、报表、推荐系统等业务上。在广告业务方面需要大数据做应用分析、效果分析、定向优化等，在推荐系统方面则需要大数据优化相关排名、个性化推荐以及热点点击分析等。这些应用场景的普遍特点是计算量大、效率要求高。

我们所指的大数据不同于过去传统的数据，其产生方式、存储载体、访问方式、表现形式、来源特点等都同传统数据不同。大数据更接近于某个群体行为数据，它是全面的数据、准确的数据、有价值的数据。

8.1 大数据未来发展趋势

随着大数据应用范围的不断扩大，越来越多的公司开始部署大数据战略。同时，大数据技术也使得商业发展的速度更快、效率更高。通过大数据技术，企业可以更轻松地获取信息，以便更准确地决策。

1. 数据量增长

数据量的不断增加意味着通过数据的快速分析获取宝贵的市场洞察已经成为大数据业务运营的关键环节。机构和企业组织必须将其内部未被利用的每一字节的大数据，也就是我们所谓的“黑暗数据（dark data）”加以合理化地整合并转化成可以利用的数据资源。

2. 大数据助推客户体验

利用大数据更深入地了解客户需求，使企业可以通过搭配销售或者促销活动提高自己的一线财政收入水平，同时还可以免除因客户流失所导致的业绩缩水风险。消费者使用灵活性的自助服务方式可以让大数据分析为企业快速掌握市场发展的主导趋势提供依据，还可以为客户需求增长机遇带来更多有竞争力的市场洞察。

3. 大数据预测分析

精准地预测未来可能发生的行为和事件可以提高企业的利润。为降低企业经营风险，暴露所面临的欺诈行为、快速鉴别和预判技术将会迎来质的飞跃，同时企业运营的卓越性将进一步得到改进。

4. 迁移到云端

将数据分析业务迁移到云端可以加速企业采用最新的技术能力，并实现数据资源到行动计划的快速转变。数据分析业务转移到云端之后，企业的运营和技术维护成本也将削减不少。

5. 分析数据价值

使用信息学助推复杂数据收集、分析与可视化技术的整合，可以从数据资源中推导出企业所需的收益来源。

6. 图形数据可视化

数据可视化技术让隐藏在大数据资源背后的真相呈现在众人面前。无论数据怎样形成，无论数据资源在哪里，图形数据可视化可以让企业组织在业务繁忙的同时对数据进行检索与处理。

7. 物联网、云技术、大数据和网络安全深度融合

数据质量控制、数据准备、数据分析以及数据整合等方面的融合程度将达到新的高度，对智能设备的依赖程度增加、互通性以及机器学习将会成为保护资产免遭网络安全危害的重要手段。

8. 偏好分析

以客户偏好的渠道与其保持有效接触，可以让企业在传统渠道与数字渠道之间找到最佳平衡点。通过不同渠道不断寻求创新手段、提高客户体验度可以带来企业的竞争优势。

8.2 大数据给人类带来的认知冲击

人类社会的发展一直都在依赖着数据，不论是各国文明的演化、农业的规划、工业的发展、军事战役及政治事件等。但是出现大数据之后，我们将会面对着海量的数据：多种维度的数据、行为的数据、情绪的数据、实时的数据，这些数据是过去没有了解到的。通过大数据计算和分析技术，人们将会得到不同的事物真相、不同的事物发展规律。依靠大数据提供的数据分析报告，人们将会发现决定一件事、判断一件事、了解一件事不再变得困难[32]。

大数据技术就像其他的技术革命一样，是从效率提升入手。大数据技术云平台的出现提升了数据处理效率。其效率的提升是几何级数增长的，过去需要几天或更多时间处理的数据，现在可能在几分钟之内就会完成。大数据的高效计算能力，为人类节省了更多的时间。

相对于过去的样本代替全体的统计方法，大数据将使用全局的数据，其统计出来的结果更为精确，更接近事物真相，帮助科学家了解事物背后的真相。大数据带来的统计结果将纠正过去人们对事物错误的认识，影响过去人类行为、社会行为的结论的准确性，给人们带来全新的认知。有利于政府、企业、科学家了解过去人类社会的各种历史行为的真正原因，大数据统计将纠正样本统计误差，为统计结论不断纠错。大数据可以让人类更加接近了解大自然，增加对自然灾害产生原因的了解。

大数据收集了全局的数据、准确的数据，通过大数据计算统计出了解事物发展过程中的真相，通过数据分析了解人类社会的发展规律、自然界发展规律。

拥有了大数据技术之后，大量的传感器如手机 APP、摄像头、分享的图片和视频等让我们更加客观地了解人类的行为。

8.3 未来大数据研究突破的技术问题

随着大数据的应用范围不断扩大，越来越多的公司开始部署大数据战略。同时，大数据技术也使得商业发展的速度更快、效率更高。通过大数据技术，企业可以更轻松地获取信息，以便更准确地决策。

1. 大数据推动新工具的出现

虽然 SQL 依然会是数据分析的标准方法，但是随着数据量的不断增长，数据分析方法也将进一步发展。大数据时代下的快速处理数据分析工作的框架，会随着应用和硬件技术的发展获得突破，目前多家世界顶级的数据企业例如 Google、Facebook 等现已纷纷转向 Spark 框架。应用与普及推动新兴分析工具趋向操作简单，对用户没有任何编码知识要求。如 Microsoft 和 Salesforce 都已经推出了新型分析工具：Microsoft R Server 和 Lightning CRM 平台，非编码人员也可以创建应用程序来查看数据。

2. 实时数据分析提升生产效率

企业需要实时数据分析工具来帮助他们利用数据进行实时决策。实时计算一般都是针对海量数据进行的，一般要求为秒级。目前有几款数据分析工具可以提供实时访问数据，如 Google Analytics 和 Clicky。

3. 数据隐私保护

从目前的信息泄露案件分析，近 60%的企业都将面临隐私泄露问题。事实上，互联网企业制定和实施新的隐私规则时，早已经预见到了这一点。大数据时代，解决用户隐私泄露问题，就是解决大数据发展与使用的问题。

4. 人工智能应用已经达到临界点

随着人工智能技术日益成熟，未来公司企业将很大程度上依赖于这项技术。无人驾驶汽车试驾成功、AlphaGo 围棋获胜。虚拟助手、机器人、智能顾问和自动驾驶汽车等多种技术

都将得到广泛地应用。

5. 大数据加速认知技术发展

随着人工智能的发展，认知技术的重要性越来越受到人们的认可。包括计算机视觉、机器学习、自然语言处理、机器人技术和语言识别技术等。应用的发展促使人们认识到大数据和分析学之间的紧密联系，发现认知计算和分析学一样，是人类社会发展不可或缺的技术。

6. 大数据量质平衡

未来大数据应用应考虑改变信息的生产、传播、加工和组织方式，进而给各个行业的创新发展带来新的驱动力，推动各个领域的彻底变革和再造。大数据将被分割成数据块，这将打破行业领域对信息流动的限制，通过对不同类型、不同领域数据的跨界集聚。

8.4 本章小结

国内的互联网企业在大数据应用和研发方面处于较好的水平，例如淘宝、百度、腾讯、新浪等。但是在大数据产品和技术服务领域却落后于国际厂商。国际主流大数据产商包括 Cloudera、HortonWorks、MapR、IBM、Oracle、EMC、Intel、SAP 和 Teradata。

由于大数据及大数据技术是一个工具，无法像互联网企业那样形成一个大数据生态圈，形成闭环。但是从数据的收集、存储、处理、分析、销毁等方面分析，可以形成大数据产业链。图 8-1 给出了未来的大数据发展图谱。



图 8-1　未来的大数据发展图谱

1. 大数据基础服务

大数据基础服务是云大数据服务的基石，解决数据的存、通问题；通过数加平台，用相同的数据标准将数据进行正确的关联，进而可以进行上层数据分析及应用。大数据基础服务

包含以下技术概念：

（1）大数据计算服务（MaxCompute）是一种快速、完全托管的 TB/PB 级数据仓库解决方案。MaxCompute 向用户提供了完善的数据导入方案以及多种经典的分布式计算模型，能够更快速地解决用户海量数据计算问题，有效降低企业成本，并保障数据安全。

（2）大数据开发套件（Data IDE），提供可视化开发界面、离线任务调度运维、快速数据集成、多人协同工作等功能，为你提供一个高效、安全的离线数据开发环境，并且拥有强大的 Open API，为数据应用开发者提供良好的再创作生态。

（3）分析型数据库（AnalyticDB），是阿里巴巴自主研发的海量数据实时高并发在线分析（Realtime OLAP）云计算服务，使得你可以在毫秒级针对千亿级数据进行即时的多维分析透视和业务探索。分析型数据库对海量数据的自由计算和极速响应能力，能让用户在瞬息之间进行灵活的数据探索，快速发现数据价值，并可直接嵌入业务系统，为终端客户提供分析服务。

（4）云流计算（Cloud StreamCompute）是运行在阿里云平台上的流式大数据分析平台，提供给用户在云上进行流式数据实时化分析工具。

2. 数据分析及展示

通过数据分析及展现产品，用户可以实现用数据来主动发现业务问题、实现现有信息的预测分析和可视化，以帮助用户更好地讲故事，帮助企业快速获得切实有效的业务见解。

（1）数据可视化专精于业务数据与地理信息融合的大数据可视化，通过图形界面轻松搭建专业的可视化应用，满足你日常业务监控、调度、会展演示等多场景使用需求。

（2）海量数据实时在线分析、拖曳式操作、丰富的可视化效果，帮助你轻松自如地完成数据分析、业务数据探查。它不只是业务人员“看”数据的工具，更是数据化运营的助推器，Data Intelligence more than Business Intelligence，实现人人都是数据分析师。

（3）画像分析将分布在多个存储资源的数据整合起来，在标签模型上构建大数据画像类的交互式分析应用，让你的业务人员可以自由灵活地分析这些对象各种属性与行为之间的关联性，可以广泛应用于用户行为、设备管理、企业档案、地理分布等多种画像分析的多个场景当中。

3. 数据应用

把用户、数据和算法巧妙地连接起来的是数据应用。数加平台提供的数据应用产品完全具备智能模块和学习功能，将助力企业颠覆传统商业。

（1）推荐引擎（Recommendation Engine，简称 RecEng）是在云计算环境下建立的一套推荐服务框架，用于实时预测用户对物品的偏好，支持你定制推荐算法，支持 A/B Test 效果对比。

（2）公众趋势分析是基于全网公开发布数据、传播路径和受众群体画像，利用语义分析、情感算法和机器学习，分析公众对品牌形象、热点事件和公共政策的认知趋势。

（3）企业图谱（Enterprise Profile，简称 E-profile）提供企业多维度信息查询，深度挖掘企业与企业、企业与个人关系链路，方便企业构建基于企业画像及企业关系网络的风险控制、市场监测等企业级服务。

（4）数据集成（Data Integration）是阿里集团对外提供的稳定高效、弹性伸缩的数据同步平台，为阿里云各个云产品（包括 MaxCompute、Analytic DB、OSS、OTS、RDS 等）提供离线（批量）数据进出通道。

4. 人工智能

大数据真正的价值在算法，算法决定行动，算法也是“机器学习”的核心，机器学习又是“人工智能”的核心。数加平台通过机器学习促成了语音、图像、视频识别等技术领域的快速发展。

（1）云机器学习是基于云分布式计算引擎的机器学习算法应用平台。用户通过拖拉拽的方式，可视化地操作组件来进行试验，使得没有机器学习背景的工程师也可以轻易上手，玩转数据挖掘。平台提供了丰富的组件，包括数据预处理、特征工程、算法组件、预测与评估。所有算法都经历了阿里云内部业务的大数据的锤炼。阿里云机器学习帮助你的业务从 BI 跨入 AI，让越来越多的人享受人工智能带来的福利。

（2）智能语音交互（Intelligent Speech Interaction）是基于语音识别、语音合成、自然语言理解等技术，为企业在多种实际应用场景下，赋予产品“能听、会说、懂你”式的智能人机交互体验。适用于多个应用场景中，包括智能问答、智能质检、法庭庭审实时记录、实时演讲字幕、访谈录音转写等场景，在金融、保险、司法、电商等多个领域均有应用案例。

（3）印刷文字识别（OCR），通俗来讲就是将图片中的文字识别出来。提供的服务包括身份证文字识别、门店招牌识别、行驶证识别、驾驶证识别、名片识别等证件类文字识别场景。

（4）人脸识别是一款用于提供图像和视频帧中人脸分析的在线服务。我们提供人脸相关技术的在线 API 服务给开发者和企业使用，包括人脸检测、人脸特征提取、人脸年龄估计和性别识别、人脸关键点定位等独立服务模块。可应用于人脸美化、人脸识别和认证、大规模人脸检索、照片管理等各种场景。

以上应用都是未来大数据的典型应用，当然不局限于这些应用。由于目前大数据产业的商业模式和盈利模式还在探索之中，大数据带来的直接收益还没有明确，目前主要的商业形式还是多数企业自身的大数据应用（例如，大数据计算平台、大数据采集和分析、数据分析报告），行业应用处于一个探索的阶段。在大数据较为集中的电信行业，并没有成立数据事业部，数据被当作资产良好地保存起来。在国外，大数据投资在 2005 年就开始了，很多高科技企业已经在大数产业链上投入巨资进行技术开发和行业应用。

大数据的发展十分快速，与目前已经飞速发展并且极具影响力的互联网一样，对于社会的各个行业来说都是一个新的技术革命，其相关技术的普及，对于科学技术上的突破都是非常显而易见的。

在不久的未来，大数据将会成为一个专门的学科，会被更多的人所熟知和了解，并且，大数据相关职业也会逐渐普及，由于大数据的普遍使用，也会催生出更多的行业岗位，数据共享会在企业层面进行扩展，从而成为产业的核心。

云计算的存在为大数据的处理提供了强有效的支撑作用，大数据的运作与云处理是不可分割的。从 2013 年开始，云计算技术和大数据处理技术就已经有效地结合，其关系也非常密切，而随着大数据时代的不断发展，两者的关系也会更加密切和契合。

附　录

Spark MLlib神经网络算法

package NN

```
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.rdd.RDD
import org.apache.spark.Logging
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.linalg.distributed.RowMatrix

import breeze.linalg.{
  Matrix => BM,
  CSCMatrix => BSM,
  DenseMatrix => BDM,
  Vector => BV,
  DenseVector => BDV,
  SparseVector => BSV,
  axpy => brzAxpy,
  svd => brzSvd
}
import breeze.numerics.{
  exp => Bexp,
  tanh => Btanh
}

import scala.collection.mutable.ArrayBuffer
import java.util.Random
import scala.math._

/**
```

```
 * label: 目标矩阵
 * nna: 神经网络每层节点的输出值,a(0),a(1),a(2)
 * error: 输出层与目标值的误差矩阵
 */
case class NNLabel(label: BDM[Double], nna: ArrayBuffer[BDM[Double]], error:
BDM[Double]) extends Serializable

/**
 * 配置参数
 */
case class NNConfig(
  size: Array[Int],
  layer: Int,
  activation_function: String,
  learningRate: Double,
  momentum: Double,
  scaling_learningRate: Double,
  weightPenaltyL2: Double,
  nonSparsityPenalty: Double,
  sparsityTarget: Double,
  inputZeroMaskedFraction: Double,
  dropoutFraction: Double,
  testing: Double,
  output_function: String) extends Serializable

/**
 * NN(neural network)
 */

class NeuralNet(
  private var size: Array[Int],
  private var layer: Int,
  private var activation_function: String,
  private var learningRate: Double,
  private var momentum: Double,
  private var scaling_learningRate: Double,
  private var weightPenaltyL2: Double,
  private var nonSparsityPenalty: Double,
  private var sparsityTarget: Double,
```

```
  private var inputZeroMaskedFraction: Double,
  private var dropoutFraction: Double,
  private var testing: Double,
  private var output_function: String) extends Serializable with Logging {
  //          var size=Array(5, 7, 1)
  //          var layer=3
  //          var activation_function="tanh_opt"
  //          var learningRate=2.0
  //          var momentum=0.5
  //          var scaling_learningRate=1.0
  //          var weightPenaltyL2=0.0
  //          var nonSparsityPenalty=0.0
  //          var sparsityTarget=0.05
  //          var inputZeroMaskedFraction=0.0
  //          var dropoutFraction=0.0
  //          var testing=0.0
  //          var output_function="sigm"
  /**
   * size = architecture;
   * n = numel(nn.size);
   * activation_function = sigm   隐含层函数 Activation functions of hidden layers:
'sigm' (sigmoid) or 'tanh_opt' (optimal tanh).
   * learningRate = 2;           学习率 learning rate Note: typically needs to be
lower when using 'sigm' activation function and non-normalized inputs.
   * momentum = 0.5;             Momentum
   * scaling_learningRate = 1;    Scaling factor for the learning rate (each
epoch)
   * weightPenaltyL2  = 0;        正则化 L2 regularization
   * nonSparsityPenalty = 0;      权重稀疏度惩罚值 on sparsity penalty
   * sparsityTarget = 0.05;       Sparsity target
   * inputZeroMaskedFraction = 0; 加入 noise,Used for Denoising AutoEncoders
   * dropoutFraction = 0;         每一次 mini-batch 样本输入训练时，随机扔掉 x%的隐含层节
点 Dropout level (http://www.cs.toronto.edu/~hinton/absps/dropout.pdf)
   * testing = 0;                Internal variable. nntest sets this to one.
   * output = 'sigm';            输出函数 output unit 'sigm' (=logistic),
'softmax' and 'linear'   *
   */
  def this() = this(NeuralNet.Architecture, 3, NeuralNet.Activation_Function,
2.0, 0.5, 1.0, 0.0, 0.0, 0.05, 0.0, 0.0, 0.0, NeuralNet.Output)
```

```
/** 设置神经网络结构. Default: [10, 5, 1]. */
def setSize(size: Array[Int]): this.type = {
  this.size = size
  this
}

/** 设置神经网络层数据. Default: 3. */
def setLayer(layer: Int): this.type = {
  this.layer = layer
  this
}

/** 设置隐含层函数. Default: sigm. */
def setActivation_function(activation_function: String): this.type = {
  this.activation_function = activation_function
  this
}

/** 设置学习率因子. Default: 2. */
def setLearningRate(learningRate: Double): this.type = {
  this.learningRate = learningRate
  this
}

/** 设置 Momentum. Default: 0.5. */
def setMomentum(momentum: Double): this.type = {
  this.momentum = momentum
  this
}

/** 设置 scaling_learningRate. Default: 1. */
def setScaling_learningRate(scaling_learningRate: Double): this.type = {
  this.scaling_learningRate = scaling_learningRate
  this
}

/** 设置正则化 L2因子. Default: 0. */
def setWeightPenaltyL2(weightPenaltyL2: Double): this.type = {
```

```
    this.weightPenaltyL2 = weightPenaltyL2
    this
  }

  /** 设置权重稀疏度惩罚因子. Default: 0. */
  def setNonSparsityPenalty(nonSparsityPenalty: Double): this.type = {
    this.nonSparsityPenalty = nonSparsityPenalty
    this
  }

  /** 设置权重稀疏度目标值. Default: 0.05. */
  def setSparsityTarget(sparsityTarget: Double): this.type = {
    this.sparsityTarget = sparsityTarget
    this
  }

  /** 设置权重加入噪声因子. Default: 0. */
  def setInputZeroMaskedFraction(inputZeroMaskedFraction: Double): this.type =
{
    this.inputZeroMaskedFraction = inputZeroMaskedFraction
    this
  }

  /** 设置权重 Dropout 因子. Default: 0. */
  def setDropoutFraction(dropoutFraction: Double): this.type = {
    this.dropoutFraction = dropoutFraction
    this
  }

  /** 设置 testing. Default: 0. */
  def setTesting(testing: Double): this.type = {
    this.testing = testing
    this
  }

  /** 设置输出函数. Default: linear. */
  def setOutput_function(output_function: String): this.type = {
    this.output_function = output_function
    this
```

```
  }

  /**
   * 运行神经网络算法.
   */
  def NNtrain(train_d: RDD[(BDM[Double], BDM[Double])], opts: Array[Double]):
NeuralNetModel = {
    val sc = train_d.sparkContext
    var initStartTime = System.currentTimeMillis()
    var initEndTime = System.currentTimeMillis()
    // 参数配置 广播配置
    var nnconfig = NNConfig(size, layer, activation_function, learningRate,
momentum, scaling_learningRate,
      weightPenaltyL2, nonSparsityPenalty, sparsityTarget,
inputZeroMaskedFraction, dropoutFraction, testing,
      output_function)
    // 初始化权重
    var nn_W = NeuralNet.InitialWeight(size)
    var nn_vW = NeuralNet.InitialWeightV(size)
    //        val tmpw = nn_W(1)
    //        for (i <- 0 to tmpw.rows -1) {
    //          for (j <- 0 to tmpw.cols - 1) {
    //            print(tmpw(i, j) + "\t")
    //          }
    //          println()
    //        }

    // 初始化每层的平均激活度 nn.p
    // average activations (for use with sparsity)
    var nn_p = NeuralNet.InitialActiveP(size)

    // 样本数据划分：训练数据、交叉检验数据
    val validation = opts(2)
    val splitW1 = Array(1.0 - validation, validation)
    val train_split1 = train_d.randomSplit(splitW1, System.nanoTime())
    val train_t = train_split1(0)
    val train_v = train_split1(1)

    // m:训练样本的数量
```

```
val m = train_t.count
// batchsize 是做 batch gradient 时候的大小
// 计算 batch 的数量
val batchsize = opts(0).toInt
val numepochs = opts(1).toInt
val numbatches = (m / batchsize).toInt
var L = Array.fill(numepochs * numbatches.toInt)(0.0)
var n = 0
var loss_train_e = Array.fill(numepochs)(0.0)
var loss_val_e = Array.fill(numepochs)(0.0)
// numepochs 是循环的次数
for (i <- 1 to numepochs) {
  initStartTime = System.currentTimeMillis()
  val splitW2 = Array.fill(numbatches)(1.0 / numbatches)
  // 根据分组权重，随机划分每组样本数据
  val bc_config = sc.broadcast(nnconfig)
  for (l <- 1 to numbatches) {
    // 权重
    val bc_nn_W = sc.broadcast(nn_W)
    val bc_nn_vW = sc.broadcast(nn_vW)

    //       println(i + "\t" + l)
    //       val tmpw0 = bc_nn_W.value(0)
    //       for (i <- 0 to tmpw0.rows - 1) {
    //         for (j <- 0 to tmpw0.cols - 1) {
    //           print(tmpw0(i, j) + "\t")
    //         }
    //         println()
    //       }
    //       val tmpw1 = bc_nn_W.value(1)
    //       for (i <- 0 to tmpw1.rows - 1) {
    //         for (j <- 0 to tmpw1.cols - 1) {
    //           print(tmpw1(i, j) + "\t")
    //         }
    //         println()
    //       }

    // 样本划分
    val train_split2 = train_t.randomSplit(splitW2, System.nanoTime())
```

```
        val batch_xy1 = train_split2(l - 1)
        //        val train_split3 = train_t.filter { f => (f._1 >= batchsize *
(l - 1) + 1) && (f._1 <= batchsize * (l)) }
        //        val batch_xy1 = train_split3.map(f => (f._2, f._3))
        // Add noise to input (for use in denoising autoencoder)
        // 加入 noise，这是 denoising autoencoder 需要使用到的部分
        // 这部分请参见《Extracting and Composing Robust Features with Denoising
Autoencoders》这篇论文
        // 具体加入的方法就是把训练样例中的一些数据调整变为0，inputZeroMaskedFraction 表
示了调整的比例
        //val randNoise = NeuralNet.RandMatrix(batch_x.numRows.toInt,
batch_x.numCols.toInt, inputZeroMaskedFraction)
        val batch_xy2 = if (bc_config.value.inputZeroMaskedFraction != 0) {
          NeuralNet.AddNoise(batch_xy1, bc_config.value.inputZeroMaskedFraction)
        } else batch_xy1

        //        val tmpxy = batch_xy2.map(f =>
(f._1.toArray,f._2.toArray)).toArray.map {f => ((new ArrayBuffer() ++ f._1) ++
f._2).toArray}
        //        for (i <- 0 to tmpxy.length - 1) {
        //          for (j <- 0 to tmpxy(i).length - 1) {
        //            print(tmpxy(i)(j) + "\t")
        //          }
        //          println()
        //        }

        // NNff 是进行前向传播
        // nn = nnff(nn, batch_x, batch_y);
        val train_nnff = NeuralNet.NNff(batch_xy2, bc_config, bc_nn_W)

        //        val tmpa0 = train_nnff.map(f => f._1.nna(0)).take(20)
        //        println("tmpa0")
        //        for (i <- 0 to 10) {
        //          for (j <- 0 to tmpa0(i).cols - 1) {
        //            print(tmpa0(i)(0, j) + "\t")
        //          }
        //          println()
        //        }
        //        val tmpa1 = train_nnff.map(f => f._1.nna(1)).take(20)
```

```
//          println("tmpa1")
//          for (i <- 0 to 10) {
//            for (j <- 0 to tmpa1(i).cols - 1) {
//              print(tmpa1(i)(0, j) + "\t")
//            }
//            println()
//          }
//          val tmpa2 = train_nnff.map(f => f._1.nna(2)).take(20)
//          println("tmpa2")
//          for (i <- 0 to 10) {
//            for (j <- 0 to tmpa2(i).cols - 1) {
//              print(tmpa2(i)(0, j) + "\t")
//            }
//            println()
//          }

// sparsity 计算，计算每层节点的平均稀疏度
nn_p = NeuralNet.ActiveP(train_nnff, bc_config, nn_p)
val bc_nn_p = sc.broadcast(nn_p)

// NNbp 是后向传播
// nn = nnbp(nn);
val train_nnbp = NeuralNet.NNbp(train_nnff, bc_config, bc_nn_W, bc_nn_p)

//          val tmpd0 = rdd5.map(f => f._2(2)).take(20)
//          println("tmpd0")
//          for (i <- 0 to 10) {
//            for (j <- 0 to tmpd0(i).cols - 1) {
//              print(tmpd0(i)(0, j) + "\t")
//            }
//            println()
//          }
//          val tmpd1 = rdd5.map(f => f._2(1)).take(20)
//          println("tmpd1")
//          for (i <- 0 to 10) {
//            for (j <- 0 to tmpd1(i).cols - 1) {
//              print(tmpd1(i)(0, j) + "\t")
//            }
//            println()
```

```
//        }
//        val tmpdw0 = rdd5.map(f => f._3(0)).take(20)
//        println("tmpdw0")
//        for (i <- 0 to 10) {
//          for (j <- 0 to tmpdw0(i).cols - 1) {
//            print(tmpdw0(i)(0, j) + "\t")
//          }
//          println()
//        }
//        val tmpdw1 = rdd5.map(f => f._3(1)).take(20)
//        println("tmpdw1")
//        for (i <- 0 to 10) {
//          for (j <- 0 to tmpdw1(i).cols - 1) {
//            print(tmpdw1(i)(0, j) + "\t")
//          }
//          println()
//        }

      // nn = NNapplygrads(nn) returns an neural network structure with
updated
      // weights and biases
      // 更新权重参数：w=w-α*[dw + λw]
      val train_nnapplygrads = NeuralNet.NNapplygrads(train_nnbp, bc_config,
bc_nn_W, bc_nn_vW)
      nn_W = train_nnapplygrads(0)
      nn_vW = train_nnapplygrads(1)

//        val tmpw2 = train_nnapplygrads(0)(0)
//        for (i <- 0 to tmpw2.rows - 1) {
//          for (j <- 0 to tmpw2.cols - 1) {
//            print(tmpw2(i, j) + "\t")
//          }
//          println()
//        }
//        val tmpw3 = train_nnapplygrads(0)(1)
//        for (i <- 0 to tmpw3.rows - 1) {
//          for (j <- 0 to tmpw3.cols - 1) {
//            print(tmpw3(i, j) + "\t")
//          }
```

```
        //          println()
        //        }

        // error and loss
        // 输出误差计算
        val loss1 = train_nnff.map(f => f._1.error)
        val (loss2, counte) = loss1.treeAggregate((0.0, 0L))(
          seqOp = (c, v) => {
            // c: (e, count), v: (m)
            val e1 = c._1
            val e2 = (v :* v).sum
            val esum = e1 + e2
            (esum, c._2 + 1)
          },
          combOp = (c1, c2) => {
            // c: (e, count)
            val e1 = c1._1
            val e2 = c2._1
            val esum = e1 + e2
            (esum, c1._2 + c2._2)
          })
        val Loss = loss2 / counte.toDouble
        L(n) = Loss * 0.5
        n = n + 1
      }
      // 计算本次迭代的训练误差及交叉检验误差
      // Full-batch train mse
      val evalconfig = NNConfig(size, layer, activation_function, learningRate,
momentum, scaling_learningRate,
        weightPenaltyL2, nonSparsityPenalty, sparsityTarget,
inputZeroMaskedFraction, dropoutFraction, 1.0,
        output_function)
      loss_train_e(i - 1) = NeuralNet.NNeval(train_t, sc.broadcast(evalconfig),
sc.broadcast(nn_W))
      if (validation > 0) loss_val_e(i - 1) = NeuralNet.NNeval(train_v,
sc.broadcast(evalconfig), sc.broadcast(nn_W))

      // 更新学习因子
      // nn.learningRate = nn.learningRate * nn.scaling_learningRate;
```

```
      nnconfig = NNConfig(size, layer, activation_function,
nnconfig.learningRate * nnconfig.scaling_learningRate, momentum,
scaling_learningRate,
        weightPenaltyL2, nonSparsityPenalty, sparsityTarget,
inputZeroMaskedFraction, dropoutFraction, testing,
        output_function)
      initEndTime = System.currentTimeMillis()

      // 打印输出结果
      printf("epoch: numepochs = %d , Took = %d seconds; Full-batch train mse =
%f, val mse = %f.\n", i, scala.math.ceil((initEndTime -
initStartTime).toDouble / 1000).toLong, loss_train_e(i - 1), loss_val_e(i - 1))
    }
    val configok = NNConfig(size, layer, activation_function, learningRate,
momentum, scaling_learningRate,
      weightPenaltyL2, nonSparsityPenalty, sparsityTarget,
inputZeroMaskedFraction, dropoutFraction, 1.0,
      output_function)
    new NeuralNetModel(configok, nn_W)
  }

}

/**
 * NN(neural network)
 */
object NeuralNet extends Serializable {

  // Initialization mode names
  val Activation_Function = "sigm"
  val Output = "linear"
  val Architecture = Array(10, 5, 1)

  /**
   * 增加随机噪声
   * 若随机值>=Fraction，值不变，否则改为0
   */
  def AddNoise(rdd: RDD[(BDM[Double], BDM[Double])], Fraction: Double):
RDD[(BDM[Double], BDM[Double])] = {
```

```
    val addNoise = rdd.map { f =>
      val features = f._2
      val a = BDM.rand[Double](features.rows, features.cols)
      val a1 = a :>= Fraction
      val d1 = a1.data.map { f => if (f == true) 1.0 else 0.0 }
      val a2 = new BDM(features.rows, features.cols, d1)
      val features2 = features :* a2
      (f._1, features2)
    }
    addNoise
  }

  /**
   * 初始化权重
   * 初始化为一个很小的、接近零的随机值
   */
  def InitialWeight2(size: Array[Int]): Array[BDM[Double]] = {
    // 初始化权重参数
    // weights and weight momentum
    // nn.W{i - 1} = (rand(nn.size(i), nn.size(i - 1)+1) - 0.5) * 2 * 4 *
sqrt(6 / (nn.size(i) + nn.size(i - 1)));
    // nn.vW{i - 1} = zeros(size(nn.W{i - 1}));
    val n = size.length
    val nn_W = ArrayBuffer[BDM[Double]]()
    val d1 = BDM((2.54631575950577, -2.72375471180638, -1.83131523622017, -
0.832303531504013, -1.28869970471936, -0.460188104184124), (-1.52091024201213,
1.81815348316090, -0.533406209340414, 1.77153723107141, -1.70376378930231,
1.95852409868481), (0.604392922735100, -0.312805008341265, 2.46338861792203, -
2.77264318419692, -2.74202474572555, 0.142284005609256), (-0.0792951314491902,
0.652983968878905, 2.35836765255640, -2.04274164893227, 1.39603060318734, -
1.68208055847319), (2.21352121948139, 1.65144527075334, -0.507588360889342, -
1.68141383648426, -0.310581480324221, 0.973756570035639), (1.48264358368951,
2.38613449604874, 2.22681802175890, -1.70428719030501, 2.44271213316363,
1.91268676272635), (-0.246256073282793, 1.34750367072394, -2.50094445126864,
0.587138926992906, -0.192365052800164, -2.71732925728203))
    nn_W += d1
    val d2 = BDM((1.25592501437006, -0.834980000207940, 2.29875024099543,
0.0194882319892158, 1.45126037957791, -0.492648144141757, -1.35365058999520, -
2.15014190874756))
```

```
    nn_W += d2
    nn_W.toArray
  }
  def InitialWeight(size: Array[Int]): Array[BDM[Double]] = {
    // 初始化权重参数
    // weights and weight momentum
    // nn.W{i - 1} = (rand(nn.size(i), nn.size(i - 1)+1) - 0.5) * 2 * 4 *
sqrt(6 / (nn.size(i) + nn.size(i - 1)));
    // nn.vW{i - 1} = zeros(size(nn.W{i - 1}));
    val n = size.length
    val nn_W = ArrayBuffer[BDM[Double]]()
    for (i <- 1 to n - 1) {
      val d1 = BDM.rand(size(i), size(i - 1) + 1)
      d1 :-= 0.5
      val f1 = 2 * 4 * sqrt(6.0 / (size(i) + size(i - 1)))
      val d2 = d1 :* f1
      //val d3 = new DenseMatrix(d2.rows, d2.cols, d2.data, d2.isTranspose)
      //val d4 = Matrices.dense(d2.rows, d2.cols, d2.data)
      nn_W += d2
    }
    nn_W.toArray
  }

  /**
   * 初始化权重 vW
   * 初始化为0
   */
  def InitialWeightV(size: Array[Int]): Array[BDM[Double]] = {
    // 初始化权重参数
    // weights and weight momentum
    // nn.vW{i - 1} = zeros(size(nn.W{i - 1}));
    val n = size.length
    val nn_vW = ArrayBuffer[BDM[Double]]()
    for (i <- 1 to n - 1) {
      val d1 = BDM.zeros[Double](size(i), size(i - 1) + 1)
      nn_vW += d1
    }
    nn_vW.toArray
  }
```

```
/**
 * 初始每一层的平均激活度
 * 初始化为0
 */
def InitialActiveP(size: Array[Int]): Array[BDM[Double]] = {
  // 初始每一层的平均激活度
  // average activations (for use with sparsity)
  // nn.p{i}     = zeros(1, nn.size(i));
  val n = size.length
  val nn_p = ArrayBuffer[BDM[Double]]()
  nn_p += BDM.zeros[Double](1, 1)
  for (i <- 1 to n - 1) {
    val d1 = BDM.zeros[Double](1, size(i))
    nn_p += d1
  }
  nn_p.toArray
}

/**
 * 随机让网络某些隐含层节点的权重不工作
 * 若随机值>=Fraction，矩阵值不变，否则改为0
 */
def DropoutWeight(matrix: BDM[Double], Fraction: Double): Array[BDM[Double]]
= {
  val aa = BDM.rand[Double](matrix.rows, matrix.cols)
  val aa1 = aa :> Fraction
  val d1 = aa1.data.map { f => if (f == true) 1.0 else 0.0 }
  val aa2 = new BDM(matrix.rows: Int, matrix.cols: Int, d1: Array[Double])
  val matrix2 = matrix :* aa2
  Array(aa2, matrix2)
}

/**
 * sigm 激活函数
 * X = 1./(1+exp(-P));
 */
def sigm(matrix: BDM[Double]): BDM[Double] = {
  val s1 = 1.0 / (Bexp(matrix * (-1.0)) + 1.0)
```

```
    s1
  }

  /**
   * tanh 激活函数
   * f=1.7159*tanh(2/3.*A);
   */
  def tanh_opt(matrix: BDM[Double]): BDM[Double] = {
    val s1 = Btanh(matrix * (2.0 / 3.0)) * 1.7159
    s1
  }

  /**
   * nnff 是进行前向传播
   * 计算神经网络中的每个节点的输出值;
   */
  def NNff(
    batch_xy2: RDD[(BDM[Double], BDM[Double])],
    bc_config: org.apache.spark.broadcast.Broadcast[NNConfig],
    bc_nn_W: org.apache.spark.broadcast.Broadcast[Array[BDM[Double]]]):
RDD[(NNLabel, Array[BDM[Double]])] = {
    // 第1层:a(1)=[1 x]
    // 增加偏置项b
    val train_data1 = batch_xy2.map { f =>
      val lable = f._1
      val features = f._2
      val nna = ArrayBuffer[BDM[Double]]()
      val Bm1 = new BDM(features.rows, 1, Array.fill(features.rows * 1)(1.0))
      val features2 = BDM.horzcat(Bm1, features)
      val error = BDM.zeros[Double](lable.rows, lable.cols)
      nna += features2
      NNLabel(lable, nna, error)
    }

    //    println("bc_size " + bc_config.value.size(0) + bc_config.value.size(1)
+ bc_config.value.size(2))
    //    println("bc_layer " + bc_config.value.layer)
    //    println("bc_activation_function " +
bc_config.value.activation_function)
```

```
//    println("bc_output_function " + bc_config.value.output_function)
//
//    println("tmpw0 ")
//    val tmpw0 = bc_nn_W.value(0)
//    for (i <- 0 to tmpw0.rows - 1) {
//      for (j <- 0 to tmpw0.cols - 1) {
//        print(tmpw0(i, j) + "\t")
//      }
//      println()
//    }

// feedforward pass
// 第2至 n-1层计算，a(i)=f(a(i-1)*w(i-1)')
//val tmp1 = train_data1.map(f => f.nna(0).data).take(1)(0)
//val tmp2 = new BDM(1, tmp1.length, tmp1)
//val nn_a = ArrayBuffer[BDM[Double]]()
//nn_a += tmp2
val train_data2 = train_data1.map { f =>
  val nn_a = f.nna
  val dropOutMask = ArrayBuffer[BDM[Double]]()
  dropOutMask += new BDM[Double](1, 1, Array(0.0))
  for (j <- 1 to bc_config.value.layer - 2) {
    // 计算每层输出
    // Calculate the unit's outputs (including the bias term)
    // nn.a{i} = sigm(nn.a{i - 1} * nn.W{i - 1}')
    // nn.a{i} = tanh_opt(nn.a{i - 1} * nn.W{i - 1}');
    val A1 = nn_a(j - 1)
    val W1 = bc_nn_W.value(j - 1)
    val aw1 = A1 * W1.t
    val nnai1 = bc_config.value.activation_function match {
      case "sigm" =>
        val aw2 = NeuralNet.sigm(aw1)
        aw2
      case "tanh_opt" =>
        val aw2 = NeuralNet.tanh_opt(aw1)
        //val aw2 = Btanh(aw1 * (2.0 / 3.0)) * 1.7159
        aw2
    }
    // dropout 计算
```

```
        // Dropout 是指在模型训练时随机让网络某些隐含层节点的权重不工作，不工作的那些节点可以
暂时认为不是网络结构的一部分
        // 但是它的权重得保留下来（只是暂时不更新而已），因为下次样本输入时它可能又得工作了
        // 参照 http://www.cnblogs.com/tornadomeet/p/3258122.html
        val dropoutai = if (bc_config.value.dropoutFraction > 0) {
          if (bc_config.value.testing == 1) {
            val nnai2 = nnai1 * (1.0 - bc_config.value.dropoutFraction)
            Array(new BDM[Double](1, 1, Array(0.0)), nnai2)
          } else {
            NeuralNet.DropoutWeight(nnai1, bc_config.value.dropoutFraction)
          }
        } else {
          val nnai2 = nnai1
          Array(new BDM[Double](1, 1, Array(0.0)), nnai2)
        }
        val nnai2 = dropoutai(1)
        dropOutMask += dropoutai(0)
        // Add the bias term
        // 增加偏置项 b
        // nn.a{i} = [ones(m,1) nn.a{i}];
        val Bm1 = BDM.ones[Double](nnai2.rows, 1)
        val nnai3 = BDM.horzcat(Bm1, nnai2)
        nn_a += nnai3
      }
      (NNLabel(f.label, nn_a, f.error), dropOutMask.toArray)
    }

    // 输出层计算
    val train_data3 = train_data2.map { f =>
      val nn_a = f._1.nna
      // nn.a{n} = sigm(nn.a{n - 1} * nn.W{n - 1}');
      // nn.a{n} = nn.a{n - 1} * nn.W{n - 1}';
      val An1 = nn_a(bc_config.value.layer - 2)
      val Wn1 = bc_nn_W.value(bc_config.value.layer - 2)
      val awn1 = An1 * Wn1.t
      val nnan1 = bc_config.value.output_function match {
        case "sigm" =>
          val awn2 = NeuralNet.sigm(awn1)
          //val awn2 = 1.0 / (Bexp(awn1 * (-1.0)) + 1.0)
```

```
        awn2
      case "linear" =>
        val awn2 = awn1
        awn2
    }
    nn_a += nnan1
    (NNLabel(f._1.label, nn_a, f._1.error), f._2)
  }

  // error and loss
  // 输出误差计算
  // nn.e = y - nn.a{n};
  // val nn_e = batch_y - nnan
  val train_data4 = train_data3.map { f =>
    val batch_y = f._1.label
    val nnan = f._1.nna(bc_config.value.layer - 1)
    val error = (batch_y - nnan)
    (NNLabel(f._1.label, f._1.nna, error), f._2)
  }
  train_data4
}

/**
 * sparsity 计算，网络稀疏度
 * 计算每个节点的平均值
 */
def ActiveP(
  train_nnff: RDD[(NNLabel, Array[BDM[Double]])],
  bc_config: org.apache.spark.broadcast.Broadcast[NNConfig],
  nn_p_old: Array[BDM[Double]]): Array[BDM[Double]] = {
  val nn_p = ArrayBuffer[BDM[Double]]()
  nn_p += BDM.zeros[Double](1, 1)
  // calculate running exponential activations for use with sparsity
  // sparsity 计算，计算 sparsity，nonSparsityPenalty 是对没达到 sparsitytarget 的参
数的惩罚系数
  for (i <- 1 to bc_config.value.layer - 1) {
    val pi1 = train_nnff.map(f => f._1.nna(i))
    val initpi = BDM.zeros[Double](1, bc_config.value.size(i))
    val (piSum, miniBatchSize) = pi1.treeAggregate((initpi, 0L))(
```

```
      seqOp = (c, v) => {
        // c: (nnasum, count), v: (nna)
        val nna1 = c._1
        val nna2 = v
        val nnasum = nna1 + nna2
        (nnasum, c._2 + 1)
      },
      combOp = (c1, c2) => {
        // c: (nnasum, count)
        val nna1 = c1._1
        val nna2 = c2._1
        val nnasum = nna1 + nna2
        (nnasum, c1._2 + c2._2)
      })
    val piAvg = piSum / miniBatchSize.toDouble
    val oldpi = nn_p_old(i)
    val newpi = (piAvg * 0.01) + (oldpi * 0.09)
    nn_p += newpi
  }
  nn_p.toArray
}

/**
 * NNbp 是后向传播
 * 计算权重的平均偏导数
 */
def NNbp(
  train_nnff: RDD[(NNLabel, Array[BDM[Double]])],
  bc_config: org.apache.spark.broadcast.Broadcast[NNConfig],
  bc_nn_W: org.apache.spark.broadcast.Broadcast[Array[BDM[Double]]],
  bc_nn_p: org.apache.spark.broadcast.Broadcast[Array[BDM[Double]]]):
Array[BDM[Double]] = {
  // 第 n 层偏导数: d(n)=-(y-a(n))*f'(z), sigmoid 函数 f'(z)表达式:f'(z)=f(z)*[1-
f(z)]
  // sigm: d{n} = - nn.e .* (nn.a{n} .* (1 - nn.a{n}));
  // {'softmax','linear'}: d{n} = - nn.e;
  val train_data5 = train_nnff.map { f =>
    val nn_a = f._1.nna
    val error = f._1.error
```

```
    val dn = ArrayBuffer[BDM[Double]]()
    val nndn = bc_config.value.output_function match {
      case "sigm" =>
        val fz = nn_a(bc_config.value.layer - 1)
        (error * (-1.0)) :* (fz :* (1.0 - fz))
      case "linear" =>
        error * (-1.0)
    }
    dn += nndn
    (f._1, f._2, dn)
  }
  // 第n-1至第2层导数：d(n)=-(w(n)*d(n+1))*f'(z)
  val train_data6 = train_data5.map { f =>
    // 假设 f(z) 是sigmoid函数 f(z)=1/[1+e^(-z)]，f'(z)表达式，f'(z)=f(z)*[1-
f(z)]
    // 假设 f(z) tanh f(z)=1.7159*tanh(2/3.*A) ，f'(z)表达式，f'(z)=1.7159 * 2/3
* (1 - 1/(1.7159)^2 * f(z).^2)
    //val di = ArrayBuffer( BDM((1.765226346140333)))
    //      val nn_a = ArrayBuffer[BDM[Double]]()
    //      val
a1=BDM((1.0,0.312605257000000,0.848582961000000,0.999014768000000,0.2783307710
00000,0.462701179000000))
    //      val a2=
BDM((1.0,0.838091550300577,0.996782915917104,0.118033012437165))
    //      val a3= BDM((2.18788852054974))
    //      nn_a += a1
    //      nn_a += a2
    //      nn_a += a3
    val nn_a = f._1.nna
    val di = f._3
    val dropout = f._2
    for (i <- bc_config.value.layer - 2 to 1) {
      // f'(z)表达式
      val nnd_act = bc_config.value.activation_function match {
        case "sigm" =>
          val d_act = nn_a(i) :* (1.0 - nn_a(i))
          d_act
        case "tanh_opt" =>
          val fz2 = (1.0 - ((nn_a(i) :* nn_a(i)) * (1.0 / (1.7159 * 1.7159))))
```

```
          val d_act = fz2 * (1.7159 * (2.0 / 3.0))
          d_act
        }
        // 稀疏度惩罚误差计算:-(t/p)+(1-t)/(1-p)
        // sparsityError = [zeros(size(nn.a{i},1),1) nn.nonSparsityPenalty * (-
nn.sparsityTarget ./ pi + (1 - nn.sparsityTarget) ./ (1 - pi))];
        val sparsityError = if (bc_config.value.nonSparsityPenalty > 0) {
          val nn_pi1 = bc_nn_p.value(i)
          val nn_pi2 = (bc_config.value.sparsityTarget / nn_pi1) * (-1.0) + (1.0
- bc_config.value.sparsityTarget) / (1.0 - nn_pi1)
          val Bm1 = new BDM(nn_pi2.rows, 1, Array.fill(nn_pi2.rows * 1)(1.0))
          val sparsity = BDM.horzcat(Bm1, nn_pi2 *
bc_config.value.nonSparsityPenalty)
          sparsity
        } else {
          val nn_pi1 = bc_nn_p.value(i)
          val sparsity = BDM.zeros[Double](nn_pi1.rows, nn_pi1.cols + 1)
          sparsity
        }
        // 导数: d(n)=-( w(n)*d(n+1)+ sparsityError )*f'(z)
        // d{i} = (d{i + 1} * nn.W{i} + sparsityError) .* d_act;
        val W1 = bc_nn_W.value(i)
        val nndi1 = if (i + 1 == bc_config.value.layer - 1) {
          //in this case in d{n} there is not the bias term to be removed
          val di1 = di(i - 1)
          val di2 = (di1 * W1 + sparsityError) :* nnd_act
          di2
        } else {
          // in this case in d{i} the bias term has to be removed
          val di1 = di(i - 1)(::, 1 to -1)
          val di2 = (di1 * W1 + sparsityError) :* nnd_act
          di2
        }
        // dropoutFraction
        val nndi2 = if (bc_config.value.dropoutFraction > 0) {
          val dropouti1 = dropout(i)
          val Bm1 = new BDM(nndi1.rows: Int, 1: Int, Array.fill(nndi1.rows *
1)(1.0))
          val dropouti2 = BDM.horzcat(Bm1, dropouti1)
```

```
        nndi1 :* dropouti2
      } else nndi1
      di += nndi2
    }
    di += BDM.zeros(1, 1)
    // 计算最终需要的偏导数值：dw(n)=(1/m)∑d(n+1)*a(n)
    //  nn.dW{i} = (d{i + 1}' * nn.a{i}) / size(d{i + 1}, 1);
    val dw = ArrayBuffer[BDM[Double]]()
    for (i <- 0 to bc_config.value.layer - 2) {
      val nndW = if (i + 1 == bc_config.value.layer - 1) {
        (di(bc_config.value.layer - 2 - i).t) * nn_a(i)
      } else {
        (di(bc_config.value.layer - 2 - i)(::, 1 to -1)).t * nn_a(i)
      }
      dw += nndW
    }
    (f._1, di, dw)
  }
  val train_data7 = train_data6.map(f => f._3)

  // Sample a subset (fraction miniBatchFraction) of the total data
  // compute and sum up the subgradients on this subset (this is one map-
reduce)
  val initgrad = ArrayBuffer[BDM[Double]]()
  for (i <- 0 to bc_config.value.layer - 2) {
    val init1 = if (i + 1 == bc_config.value.layer - 1) {
      BDM.zeros[Double](bc_config.value.size(i + 1), bc_config.value.size(i) +
1)
    } else {
      BDM.zeros[Double](bc_config.value.size(i + 1), bc_config.value.size(i) +
1)
    }
    initgrad += init1
  }
  val (gradientSum, miniBatchSize) = train_data7.treeAggregate((initgrad,
0L))(
    seqOp = (c, v) => {
      // c: (grad, count), v: (grad)
      val grad1 = c._1
```

```
    val grad2 = v
    val sumgrad = ArrayBuffer[BDM[Double]]()
    for (i <- 0 to bc_config.value.layer - 2) {
      val Bm1 = grad1(i)
      val Bm2 = grad2(i)
      val Bmsum = Bm1 + Bm2
      sumgrad += Bmsum
    }
    (sumgrad, c._2 + 1)
  },
  combOp = (c1, c2) => {
    // c: (grad, count)
    val grad1 = c1._1
    val grad2 = c2._1
    val sumgrad = ArrayBuffer[BDM[Double]]()
    for (i <- 0 to bc_config.value.layer - 2) {
      val Bm1 = grad1(i)
      val Bm2 = grad2(i)
      val Bmsum = Bm1 + Bm2
      sumgrad += Bmsum
    }
    (sumgrad, c1._2 + c2._2)
  })
 // 求平均值
 val gradientAvg = ArrayBuffer[BDM[Double]]()
 for (i <- 0 to bc_config.value.layer - 2) {
   val Bm1 = gradientSum(i)
   val Bmavg = Bm1 :/ miniBatchSize.toDouble
   gradientAvg += Bmavg
 }
 gradientAvg.toArray
}

/**
 * NNapplygrads 是权重更新
 * 权重更新
 */
def NNapplygrads(
  train_nnbp: Array[BDM[Double]],
```

```
    bc_config: org.apache.spark.broadcast.Broadcast[NNConfig],
    bc_nn_W: org.apache.spark.broadcast.Broadcast[Array[BDM[Double]]],
    bc_nn_vW: org.apache.spark.broadcast.Broadcast[Array[BDM[Double]]]):
Array[Array[BDM[Double]]] = {
    // nn = nnapplygrads(nn) returns an neural network structure with updated
    // weights and biases
    // 更新权重参数：w=w-α*[dw + λw]
    val W_a = ArrayBuffer[BDM[Double]]()
    val vW_a = ArrayBuffer[BDM[Double]]()
    for (i <- 0 to bc_config.value.layer - 2) {
      val nndwi = if (bc_config.value.weightPenaltyL2 > 0) {
        val dwi = train_nnbp(i)
        val zeros = BDM.zeros[Double](dwi.rows, 1)
        val l2 = BDM.horzcat(zeros, dwi(::, 1 to -1))
        val dwi2 = dwi + (l2 * bc_config.value.weightPenaltyL2)
        dwi2
      } else {
        val dwi = train_nnbp(i)
        dwi
      }
      val nndwi2 = nndwi :* bc_config.value.learningRate
      val nndwi3 = if (bc_config.value.momentum > 0) {
        val vwi = bc_nn_vW.value(i)
        val dw3 = nndwi2 + (vwi * bc_config.value.momentum)
        dw3
      } else {
        nndwi2
      }
      // nn.W{i} = nn.W{i} - dW;
      W_a += (bc_nn_W.value(i) - nndwi3)
      // nn.vW{i} = nn.momentum*nn.vW{i} + dW;
      val nnvwi1 = if (bc_config.value.momentum > 0) {
        val vwi = bc_nn_vW.value(i)
        val vw3 = nndwi2 + (vwi * bc_config.value.momentum)
        vw3
      } else {
        bc_nn_vW.value(i)
      }
      vW_a += nnvwi1
```

```
    }
    Array(W_a.toArray, vW_a.toArray)
  }

  /**
   * nneval 是进行前向传播并计算输出误差
   * 计算神经网络中的每个节点的输出值，并计算平均误差;
   */
  def NNeval(
    batch_xy: RDD[(BDM[Double], BDM[Double])],
    bc_config: org.apache.spark.broadcast.Broadcast[NNConfig],
    bc_nn_W: org.apache.spark.broadcast.Broadcast[Array[BDM[Double]]]): Double
= {
    // NNff 是进行前向传播
    // nn = nnff(nn, batch_x, batch_y);
    val train_nnff = NeuralNet.NNff(batch_xy, bc_config, bc_nn_W)
    // error and loss
    // 输出误差计算
    val loss1 = train_nnff.map(f => f._1.error)
    val (loss2, counte) = loss1.treeAggregate((0.0, 0L))(
      seqOp = (c, v) => {
        // c: (e, count), v: (m)
        val e1 = c._1
        val e2 = (v :* v).sum
        val esum = e1 + e2
        (esum, c._2 + 1)
      },
      combOp = (c1, c2) => {
        // c: (e, count)
        val e1 = c1._1
        val e2 = c2._1
        val esum = e1 + e2
        (esum, c1._2 + c2._2)
      })
    val Loss = loss2 / counte.toDouble
    Loss * 0.5
  }
}
```

参考文献

[1] 程学旗，靳小龙，王元卓等.大数据系统和分析技术综述[J].软件报，2014，25(9)：1889~1908

[2] http://www.sequoiadb.com/cn

[3] 黄宜华，苗凯翔.深入理解大数据——大数据处理与编程实践[M].北京:机械工业出版社，2014

[4] 林子雨，大数据技术原理与应用：概念、存储、处理、分析与应用[M].北京:人民邮电出版社，2015

[5] http://www.china-cloud.com/

[6] 杨青峰.智慧的维度——工业 4.0 时代的智慧制造[M].北京:电子工业出版社, 2014

[7] Jothy R，Arthur M，胡键译.云计算揭秘—企业实施云计算的核心问题[M].北京:机械工业出版社, 2012

[8] 杨青峰.信息化 2.0+:云计算时代的信息化体系[M].北京:电子工业出版社, 2013

[9] 刘鹏.云计算[M].北京:电子工业出版社, 2011

[10] S. P. T. Krishnan,Jose L, Ugia Gonzalez.Building Your Next Big Thing with Google Cloud Platform[M]. Apress,2015, 185~210

[11] 马友忠，孟小峰.云数据管理索引技术研究[J] .软件学报,2015,26(01):145~166

[12] 史英杰，孟小峰.云数据管理系统中查询技术研究综述[J].软件学报，2013,36(2)：209~225

[13] 王丽丽.云计算中虚拟化技术的安全问题及对策研究[J]. 首都师范大学学报（自然科学版），2015，36(4)

[14] Chirag Modi,Dhiren Patel,Bhavesh Borisaniya,Avi Patel,Muttukrishnan Rajarajan. A survey on security issues and solutions at different layers of Cloud computing[J]. The Journal of Supercomputing . 2013,(2)

[15] Oma Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster. Deconstructing Amazon EC2S-pot Instance Pricing [C]. In: 2011 Third IEEE International Conference on Coud Computing Technology and Science. USA: IEEEPress,2011.304～311

[16] Ramakrishnan L, Murki K,Canon S.PerformanceAnalysis of High performance Computing Applications on the Amazon Web Services Cloud[C]. In: 2010 IEEE Second International Conference. USA: Cloud Com, 2010.159～168

[17] 赵立威,方国伟.让云触手可及微软云计算实践指南[M].北京:电子工业出版社,2011.26～28

[18] MichaelCusumano.Cloudcomputing and SaaS as new computing platforms[M].

Communica-tions of the ACM, 2010, 53(4): 27～29

[19] Dillon. Cloud Computing: Issues and Challenges [C]. In: 2010 24th IEEE International Confe-rence on Advanced Information Networking and Applications. USA: IEEE Press,2010.27～33

[20] Sushil Bhardwaj, Leena Jain, Sandeep Jain. Cloud Computing: A Study of Infrastructure as aService[J].International Journal of Engineering andInformation Technology,2010, 2(1): 60～63

[21] M.Rosenblum,T. Garfinkel. Virtual Machine Monitors:Current Technology and Future Trends [J].IEEE Computer Society,May 2005,Volume: 30, issue: 7:39～47

[22] Mathew L Massie, Brent N Chun, David E Culler. The ganglia distributed monitoring system:design, implementation, and experience [J]. Parallel Computing, 2004, 30(1): 817～840

[23] 陈熠. 大规模机群监控系统的研究与实现 [D]. 北京:中国科学院研究生院, 2004

[24] M.Rosenblum,T.Garfinkel.VirtualMachine Monitors: Current Technology and Future Trends[J].IEEE Computer Society, May 2005,Volume:30,issue:7:39～47

[25] Mathew L Massie, Brent N Chun, David E Culler. The ganglia distributed monitoring system:design, implementation, and experience [J]. Parallel Computing, 2004, 30(1): 817～840

[26] Emir Imamagic, Dobrisa Dobrenic. Grid infrastructure monitoring system based on Nagios[C] .In:Proceedings of the 2007 workshop on Grid monitoring. USA: ACM Press, 2007. 23～28

[27] https://chukwa.apache.org/

[28]MapReduce : Simplified Data Processing on Large Clusters. In proceedings of OSDI'04

[29] Wikipedia. http://en.wikipedia.org/wiki/Mapreduce

[30] Phoenix. http://mapreduce.stanford.edu/

[31] Dean, Jeffrey&Ghemawat, Sanjay (2004). MapReduce: Simplified Data Processing on Large Clusters. Retrieved Apr. 6, 2005

[32] 36dsj.com

[33] Agrawal R, Srikant R. Privacy preserving data mining[C]//Proc. Of SIGMOD 2000. NEW York: ACM, 2000,439~450

[34] Wang C, Ren K, Yu S, et al. Achieving usable and privacy-assured similarity search over out sourced cloud data[C]// INFOCOM,2012 Preceedings IEEE. IEEE,2012,451-459

[35] 杨辅祥,刘云超,段智华等.数据清理综述[J].计算机应用研究,2002,19(3):3~5

[36] https://www.aliyun.com/

[37]Latest release spark documentation[EB/OL].http://spark.apache.org/docs/latest/,2015

[38] Carlini E,Dazzi P,Esposito A,et a1.Balanced Graph Partitioning with Apache Spark[J].Lecture Notes in Computer Science,2014

[39] 高彦杰. Spark 大数据处理：技术、应用与性能优化[M].北京:机械工业出版社，2014

[40] http://spark.apache.org

[41] http://en.wikipedia.org/wiki/Apache_Spark

[42] http://ampcamp.berkeley.edu/exercises-strata-conf-2013/launching-a-cluster.html

[43] https://spark.apache.org/docs/1.0.1/quick-start.html

[44] Kiejin Park, Changwon Baek, Limei Peng.A Development of Streaming Big Data Analysis System Using In-memory Cluster Computing Framework: Spark[C]//. Advanced Multimedia and

UbiquitousEngineering pp 157~163

[45] Choi H, Lee KY (2014) Efficient processing of an aggregate query stream in MapReduce[C]//. KIPS,Trans Softw Data Eng 3(2):73–80

[46] Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I (2013) Discretized streams: fault-tolerant streaming computation at scale[C]//, ACM, pp 423–438

[47] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin M, Shenker S, Stoica I Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing[C]//, (2012) NSDI

[48] http://www.cehuajie.cn/a/yingxiao/lilun/2015/0523/858.html

[49]Mohammed Guller,Big Data Analytics with Spark[M], 2015, pp 103-152

[50] Xinhui Tian,Gang Lu,Xiexuan Zhou,Jingwei Li.Evolution from Shark to Spark SQL: Preliminary Analysis and Qualitative Evaluation.Big Data Benchmarks,Performance Optimization,andEmerging Hardware pp 67~80

[51] Spark SQL: Relational Data Processing in Spark. Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia. SIGMOD 2015. June 2015

[52] https://dzone.com/refcardz/apache-spark

[53] http://www.cnblogs.com/BYRans/

[54] http://www.ctocio.com/hotnews/15919.html

[55] https://spark.apache.org/docs/latest/mllib-guide.html

[56] 黄美灵.Spark MLlib 机器学习-算法、源码及实战详解[M].电子工业出版社,2016

[57] MLlib: Machine Learning in Apache Spark, Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Journal of Machine Learning Research (JMLR). 2016

[58] SparkR: Scaling R Programs with Spark, Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael Franklin, Ion Stoica, and Matei Zaharia. SIGMOD 2016. June 2016

[59] GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. Reynold S. Xin, Daniel Crankshaw, Ankur Dave, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica. OSDI 2014. October 2014

[60] Discretized Streams: Fault-Tolerant Streaming Computation at Scale. Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica. SOSP 2013. November 2013

[61] Shark: SQL and Rich Analytics at Scale. Reynold S. Xin, Joshua Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, Ion Stoica. SIGMOD 2013. June 2013

Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica. HotCloud 2012. June 2012

[62] Shark: Fast Data Analysis Using Coarse-grained Distributed Memory (demo). Cliff Engle,

Antonio Lupher, Reynold S. Xin, Matei Zaharia, Haoyuan Li, Scott Shenker, Ion Stoica. SIGMOD 2012. May 2012. Best Demo Award

[63] Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. NSDI 2012. April 2012. Best Paper Award

[64] Spark: Cluster Computing with Working Sets. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. HotCloud 2010. June 2010

[65] PrivateClean: Data Cleaning and Differential PrivacySanjay Krishnan, Jiannan Wang, Michael Franklin, Ken Goldberg, Tim KraskaSIGMOD, Jun. 2016

[66] Mohammed Guller, Machine Learning with Spark,Chapter OFBig Data Analytics with Spark,2015,pp 153~205

[67] Salman Salloum,Ruslan Dautov,Xiaojun Chen, Patrick Xiaogang Peng,Joshua ZhexueHuang. Big data analytics on Apache Spark.November 2016, Volume 1, Issue 3, pp 145~164

[68] Zubair Nabi. Machine Learning at Scale,Chapter of Pro Spark Streaming,2016,pp 177-198.

[69] Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., Stoica, I.: Blinkdb: queries with bounded errors and bounded response times on very large data. In: Proceedings of the 8th ACM European Conference on Computer Systems. ACM, New York, pp.29~42 (2013). doi:10.1145/2465351.2465355

[70] Amde, M., Bradley, J.: Scalable decision trees in mllib.https://databricks.com/blog/2014/09/29/scalable-decision-trees-in-mllib.html (2014)

[71] Anagnostopoulos, I., Zeadally, S., Exposito, E.: Handling big data: research challenges and future directions. J. Supercomput. (2016). doi:10.1007/s11227-016-1677-zGoogle Scholar

[72] Andrew, G., Gao, J.: Scalable training of l1-regularized log-linear models. In: International Conference on Machine Learning (2007)

[73] Apiletti, D., Garza, P., Pulvirenti, F.: New Trends in databases and information systems: ADBIS 2015 Short Papers and Workshops, BigDap, DCSA, GID, MEBIS, OAIS, SW4CH, WISARD, Poitiers, France, September 8–11, 2015. Proceedings, Springer International Publishing, Cham, chap A Review of Scalable Approaches for Frequent Itemset Mining, pp. 243–247 (2015)

[74] Aridhi, S., Nguifo, E.M.: Big graph mining: frameworks and techniques. arXiv preprintarXiv:1602.03072 (2016)

[75] Armbrust, M., Ghodsi, A., Zaharia, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J.: Spark SQL. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data—SIGMOD '15, ACM Press, New York, NY, USA, pp. 1383~1394. doi:10.1145/2723372.2742797.http://dl.acm.org/citation.cfm?id=2723372.2742797 (2015)

[76] Armbrust, M., Huai, Y., Liang, C., Xin, R., Zaharia, M.: Deep dive into spark sqls catalyst optimizer. https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html (2015)

[77] Armbrust, M., Fan, W., Xin, R., Zaharia, M.: Introducing spark datasets.https://databricks.com/blog/2016/01/04/introducing-spark-datasets.html (2016)

[78] Awan, A.J., Brorsson, M., Vlassov, V., Ayguadé, E.: How data volume affects spark based data analytics on a scale-up server. CoRR arxiv:1507.08340 (2015)

[79] Awan, A.J., Brorsson, M., Vlassov, V., Ayguadé, E.: Architectural impact on performance of in-memory data analytics: Apache spark case study. CoRR arXiv:1604.08484 (2016)

[80] Boehm, M., Tatikonda, S., Reinwald, B., Sen, P., Tian, Y., Burdick, D.R., Vaithyanathan, S.: Hybrid parallelization strategies for large-scale machine learning in systemML. Proc. VLDB Endow. **7**(7), pp.553~564 (2014). doi:10.14778/2732286.2732292CrossRefGoogle Scholar

[81] Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: Haloop: efficient iterative data processing on large clusters. Proc. VLDB Endow. **3**(1–2), pp.285~296 (2010). doi:10.14778/1920841.1920881 CrossRefGoogle Scholar

[82] Burdorf, C.: Use of spark mllib for predicting the offlining of digital media. Presentation.https://spark-summit.org/2015/events/use-of-spark-mllib-for-predicting-the-offlining-of-digital-media/ (2015)

[83] Balcan, M.-F.F., Ehrlich, S., Liang, Y.: Distributed k-means and k-median clustering on general topologies. In: Burges, C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, k. (eds.) Advances in Neural Information Processing Systems 26, pp.1995–2003. Curran Associates Inc. (2013)

[84] Abdelkader, M., Abd-Almageed, W., Srivastava, A., Chellappa, R.: Silhouette-based gesture and action recognition via modeling trajectories on riemannian shape manifolds. Comput. Vis. Image Underst. **115**(3), pp.439–455 (2011)

[85] Al-Jarrah, O.Y., Yoo, P.D., Muhaidat, S., Karagiannidis, G.K., Taha, K.: Efficient machine learning for big data: a review. Big Data Res. **2**(3), pp.87–93 (2015)

[86] Cane, J., O'Connor, D., Michie, S.: Validation of the theoretical domains framework for use in behaviour change and implementation research. Implementation Sci. **7**(1), 1 (2012)

[87] Dakshi Agrawal, Ali Butt, Kshitij Doshi, Josep-L. Larriba-Pey, Min Li. SparkBench – A Spark Performance Testing Suite. TPCTC 2015: Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things pp 26~44

[88] DataBricks. https://databricks.com/

[89] Mahout. http://mahout.apache.org/

[90] Sara Landset, Taghi M. Khoshgoftaar, Aaron N. Richter. A survey of open source tools for machine learning with big data in the Hadoop ecosystem. Journal of Big Data (2015) 2: 24. doi:10.1186/s40537-015-0032-1

[91] International Data Corporation. Digital Universe Study. 2014

[92] The R Project for Statistical Computing. http://www.r-project.org/

[93] Weka. http://www.cs.waikato.ac.nz/ml/weka/

[94] Apache Hadoop. https://hadoop.apache.org/

[95] Supriya B.N., Prakash S., Akki C.B. (2016) A Survey on Big Data Architectures and Standard Bodies. In: Satapathy S., Bhatt Y., Joshi A., Mishra D. (eds) Proceedings of the International Congress on Information and Communication Technology. Advances in Intelligent

Systems and Computing, vol 438. Springer, Singapore

[96] http://en.wikipedia.org/wiki/Sessionization

[97] http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html

[98] Raoul Biagioni, Sentiment Analysis,Chapter ofThe SenticNet Sentiment Lexicon: Exploring Semantic Richness in Multi-Word Concepts,Volume 4 of the series SpringerBriefs in Cognitive Computation 2016,pp 7~16

[99] Alpaydin E (2010) Introduction to machine learning, 2nd edn. MIT Press, Cambridge, Mas

[100] Bethard S, Yu H, Thornton A, Hatzivassiloglou V, Jurafsky D (2004) Automatic extraction of opinion propositions and their holders. In: In 2004 AAAI spring symposium on exploring attitude and affect in text, pp. 22~24

[101] 高彦杰.Spark 大数据处理：技术、应用与性能优化[M].北京:机械工业出版社,2014

[102] M. Abirami, V. Pattabiraman. Data Mining Approach for Intelligent Customer Behavior Analysis for a Retail Store[C]//.Proceedings of the 3rd International Symposium on Big Data and Cloud Computing Challenges (ISBCC – 16') pp 283~291

[103] Decision of the European Court of Justice (Second Chamber) 11 July 2013 – Cases No. C-521/11, "Amazon",IIC - International Review of Intellectual Property and Competition Law,September 2013, Volume 44, Issue 6, pp 726~727

[104] Teijeiro D., Pardo X.C., González P., Banga J.R., Doallo R. (2016) Implementing Parallel Differential Evolution on Spark. In: Squillero G., Burelli P. (eds) Applications of Evolutionary Computation. EvoApplications 2016. Lecture Notes in Computer Science, vol 9598. Springer, Cham.

[105] 夏俊鸾,刘旭晖等.Spark 大数据处理技术[M].北京:电子工业出版社,2015

[106] Zubair Nabi, Introduction to Spark,chapter of Pro Spark Streaming, 14 June 2016, pp 9~27

[107] https://www.qcloud.com/solution/bigdata

[108] https://www.aliyun.com/product/emapreduce